个性化你的阅读

编程狂人

Programming Maniac

NO.3

推酷

关于推酷

推酷是专注于 IT 圈的个性化阅读社区。我们利用智能算法，从海量文章资讯中挖掘出高质量的内容，并通过分析用户的阅读偏好，准实时推荐给你最感兴趣的内容。我们推荐的内容包含科技、创业、设计、技术、营销等多方面内容，满足你日常的专业阅读需要。我们针对 IT 人还做了个活动频道，它聚合了 IT 圈最新最全的线上线下活动，使 IT 人能更方便地找到感兴趣的活动信息。

关于周刊

推酷周刊是专为 IT 人打造的行业技术周刊，目前推出的《编程狂人》是献给广大的程序员们。 我们利用技术挖掘出那些高质量的文章，并通过人工加以筛选整理出来。 每期的周刊一般会周一发布。

联系我们


tuicool2012


164644910

推酷网

下载 APP

Android版本

立即下载 Android

iPhone版本

免费下载就在 App Store

2013/12/9/第三期

目录

目录

后端架构

程序人生

PayPal 从 Java 切换到 JavaScript

已经决定使用 JavaScript 开发 Web 应用程序，从浏览器一直到后端服务器，并放弃了使用 JSP/Java 编写的遗留代码。

PayPal 技术总监 Jeff Harrell 在两篇博文中（解放我的 UI 第一部分：Dust JavaScript 模板、开源等、PayPal 的 Node.js）解释了他们做出这一决定的原因，并对 Web 应用程序开发从 Java/JSP 切换到完全的 JavaScript/Node.js 技术栈的过程中所产生的若干结论进行了说明。
据 Harrell 说，PayPal 的网站已经积累了大量的技术债务，他们想要一种“可以使他们摆脱债务而又能带来更大产品灵活性和创新的技术栈”。最初，在使用 Web 技术的前端工程师和使用 Java 编码的后端工程师之间存在着巨大的鸿沟。当用户体验设计人员想草绘一些页面时，为了使它们运行，他们不得不要求 Java 程序员做一些后端编码。这与他们的精益用户体验开发模型不相符：

> 当时，我们的 UI 应用程序基于 Java 和 JSP，使用了一个无弹性、紧耦合而又难以快速行动的专有解决方案。我们的团队发现它与精益用户体验开发模型不相符，而且无法快速行动，因此，他们用脚本语言构建原型，与用户一起测试，然后将代码移植到产品栈中。

他们想要一个“从底层服务器技术解耦并能使 UI 设计独立于应用程序语言的模板[解决方案]”，而且它可以工作在多种环境中。他们决定选用 Dust.js——一个由 LinkedIn 支持的模板框架——，再加上 Twitter 的 Bootstrap 和 Bower，后者是一个面向 Web 的包管理器。之后又加入了 LESS、RequireJS、Backbone.js、Grunt 和 Mocha 等其它部分。
PayPal 的部分页面已经经过重新设计，但他们仍然还有部分页面使用遗留技术栈：

> ……我们有 C++/XSL 和 Java/JSP 两个遗留技术栈，随着继续推进，我们不打算留下这些 UI。JavaScript 模板是理想之选。在 C++技术栈上，我们构建了一个使用 V8 引擎执行 Dust 本地渲染的库——其速度惊人的快！在 Java 端，我们使用 Spring ViewResolver 和 Rhino 集成 Dust 来渲染页面。

当时，他们还开始用 Node.js 进行新页面的原型设计，并认为它“非常巧妙”，进而决定在生产环境对其进行试用。为了达到这一目的，他们还构建了 Kraken.js，这是一个位于以 Node.js 为基础的 Web 框架 Express 之上的 “约定层”。（PayPal 最近开源了 Kraken.js。）第一个使用 Node.js 完成的应用程序是账户概览页，据 Harrell 说，该页面是 PayPal 的一个最经常访问页面。但是，由于担心 Node.js 应用程序可能扩展性不好，他们决定创建一个等效的 Java 应用程序，一旦 Node.js 应用程序不能正常运行，就可以回退到 Java 应用程序。
下面是关于两种应用程序开发所需工作量的几个结论：

	Java/Spring	JavaScript/Node.js
设置时间	0	2 个月

开发	大约 5 个月	大约 3 个月
工程师	5	2
代码行数	未说明	未说明，占 Java 应用程序的 66%

JavaScript 团队需要两个月的时间进行基础设施的初始设置，但他们创建具有相同功能的应用程序用人更少、耗时更短。他们在生产环境硬件上运行测试套件，得出的结论是 Node.js 应用程序的性能要优于 Java 应用程序：

> Node.js 应用程序每秒服务的请求数是 Java 应用程序的两倍。更为有趣的是，在最初的性能结果的产生过程中，Node.js 应用程序使用了单核，而 Java 应用程序使用了五核。我们打算进一步增大这种差异。

而且：

> 对于同一页面，Node.js 应用程序的平均响应时间减少了 35%。这使得页面提供时间快了 200 毫秒——用户可以明显地感觉到这种变化。

结果，PayPal 开始在生产环境中使用了尚处于测试阶段的 Node.js 应用程序，并决定“今后所有面向客户的 Web 应用程序均基于 Node.js 构建，”，而现有的部分应用程序也将迁移到 Node.js。

据 Harrell 说，从浏览器到服务器都使用 JavaScript 的一个好处是，形成了一个“允许我们在技术栈的任何层次理解和响应用户需求”的团队，消除了前端开发与后端开发之间的鸿沟。

原文：
http://www.infoq.com/cn/news/2013/12/paypal-java-javascript?utm_source=Tuicool_Weekly

疑 Google 员工把 8w 行 Python 项目用 4w 行 Java 重写了

在噩梦般地维护了一年多一个 8 万多行的 Python 程序之后，终于争取到机会把这个破烂玩意用 Java 重写了一遍，大概是 4 万行 Java 左右。说说效果吧：

1. 从过去平均每周 down 一次，到现在连续运转近半年只 down 过一次。
2. 节省超过 80%的 cpu 和内存
3. 代码多了很多功能，过去无数因为系统太复杂无法实现的功能现在都能简单清爽地实现了。
4. 单元测试真管用了，不是过去那种把代码反过来写一遍的滥用 mock 了。

前后代码都是同一个 team 写的，写代码的人都不是菜鸟（顶级公司的核心团队）区别只有语言和几年的经验积累。

总结一句话就是：动态语言滥用起来真是可怕

珍惜生命，远离 Python。

刚开始写的时候以为就是随便 hack 一个小系统临时用用，结果慢慢发展到成为关键系统，负载巨大，而且还对宕机越来越敏感，导致不得不用 Java 重写。重写也不是那么简单的过程，半年多时间里面一个模块一个模块地替换，整个系统还不能停转一分钟，像是给一架飞行中的飞机换引擎。

同样的故事在别的公司肯定也发生过好多次，写一个小东西玩玩结果变成了关键系统。我觉得我们团队的问题是几年前过分迷信 Python，错过了在系统还不太复杂的时候重写的机会。

原帖说的是一个 10 个人团队的故事，不是他自己。信与不信其实并不太重要，我发这个帖子也是纪念一下这个美梦成真的项目。如果有人有共鸣，那就已经很好了。

过去公司里面也是有不少迷信 Python 的人，重写系统的想法我在组里提了很多次，终于在一个 Python 大粉丝离开之后才得以实现。现在随着某 Python 之父的离开，公司里面粉 Python 的人也越来越少了。而且事实证明，那个 Python 之父带的项目（不是我们这个，比我们这个规模要大一些），用 Java 重写之后，不论功能还是性能还是新功能，也都明显好了很多，和我们组的经验相当吻合。

我觉得代码行数的节省也在于新系统更严谨的设计。Java 鼓励精密的接口设计和简洁的代码关系，再加上 Dependency Injection，代码的复用程度很高。Python 完全没有接口的概念，一切类都是胡乱写，还可以动态增加新成员，导致代码复用的难度相当大，不修改地复用一个类还不引入 bug 简直是奇迹。

这个项目里面，重构是一直在进行的，在过去的八年里面，用 Python 大规模地重构也发生过好几次。

决定用 Java 重写是很艰难的，不是某个人拍脑袋的决定。放弃 Python 主要是因为它的代码可维护性比较差，缺乏编译器检查，动态语言特性导致模块间过渡耦合，缺乏性能分析工具，内存管理机制低效，多线程性能很差。这些都是一个持续维护很多年的高负载应用所必须的。

Linter 能做的事情很有限，公司代码管理系统早就是不过 lint 不能 submit。code review 也不能解决问题，因为团队里面每一个人对于系统设计和语言特性是否滥用理解程度都不同。

争论是否能够通过管理来解决语言本身缺陷其实是没意义的，因为我们碰见的是实实在在的语言缺陷问题，就应该用最直接的解决方案。

“Python 很好，只是你们这些外行不会用”，这是我听到的最多的所谓 Pythonic 人的论调。事实是 Python 作者自己亲自带领的那个项目也是一塌糊涂。他们遇到的最主要问题就是可维护性低和性能差。世界上有没有人比 Python 作者本人更 Pythonic?如果他自己的项目都不行，谁能做得更好？

如果一个语言比另外一个差 1000 倍，那就是很大的问题了，随便写两行 code 都可能成为性能瓶颈。Python 又恰好缺乏性能分析和优化工具，导致出了性能问题都不知道在哪儿。BTW，1000 倍这个数据是有依据的。

接口上所谓的冗余其实是提高程序可维护性的关键。人脑容量有限，所以必须用电脑辅助人脑进行接口使用正确性的检查。强制要求程序员写强类型的接口也是一种强迫程序员对正确的设计进行思考的过程。结果导致的就是强类型语言可维护性更好。Openstack 只有 1.4M 和 3 年历史，用 c/c++/Java 写的大型软件都是复杂几个数量级，而且维护几十年。Python 程序的问题之一就是“腐烂”得很快，程序刚写好的时候看起来很简洁，一旦开始维护就快速地变成一堆乱七八糟的东西，动态语言代码腐烂速度远远超过强类型语言。
如果 Python 作者本人带领的团队都写不出可维护的 Python 代码，谁能？

Guido 搞的那个 code review 系统就是我说的那个维护不了的系统，最后的结果就是用 Java 重写了。新系统比旧系统好用很多，而且还持续有新功能上线。那个项目组有一个在公司内部流传很广的文章，说为什么需要用 Java 重写，并且结论是 Python 只适合写 100 行以内的小脚本。我刚才试了找这个文章有没有外网可以访问的版本，但是没找到。

如果用一个语言开发出可维护的代码需要的能力超过了 Google 能招聘到程序员的平均水平，我认为这个语言就不是一个可以用来开发可维护代码的语言。创造出一个不可维护的语言很容易，汇编，Basic 以及大批的早期编程语言都是这种。创造出可以让普通程序员也可以维护的语言才是更困难的，只有少数语言做到了，C 和 Java 是，C++都不能算是很成功。

你几乎都说反了，Google 非常重视代码的可维护性，扩展性几乎是在项目一开始就必须考虑的问题，Google 对软件工程实践的重视和深度是我所知道的公司里面最高的。不知道你说的管理是什么意思，但是 Google 有强制性的代码规范，有强制性的 designreview。Python 的很多“高级”特性已经在编程规范里面禁用了，但是有些特性是没法禁用的，比方所有对象 member 都是 public，比方函数参数没有类型。就是这些很基本的东西带来的可维护性问题。

越是水平高的程序员越是会遵守编程规范和软件工程实践。至少在我们这种产品 team 里面大家是相当重视代码质量和设计质量的。这也是为什么我们舍得投入那么多把一个不好的系统重写。

【 在 Alassius (饿了索食) 的大作中提到: 】

问题是这种不可维护性恐怕正是贵司能招聘到的这种平均水平造成的。智商高，技术至上，写代码怎么聪明怎么来，不重视维护，不重视扩展性（直到太迟的时

候），不重视软件工程，不重视管理。普通程序员如果有好的资深码农引导，早早地被告知不要滥用语言的灵活性，倒未必会有这些问题。聪明的程序员你这样跟他说他还看不起你。

意思就是：任何流程，任何规范，任何 review，任何代码检查工具，都阻止不了程序员缓慢地一点一点地滥用语言特性，也阻止不了一个软件代码自身的腐败。Python 本身鼓励滥用 Mock 因为没有不用 mock 就可以方便写 unit test 的测试工具。Python 还鼓励不定义清晰的类接口和类关系，如果你写了一个接口继承，就会有 Python 大牛跳出来说你不 Pythonic。系统复杂性在大型软件中是不可避免的，但是如果我们可以付出一些写 code 的时候的小小冗余，带来长期的代码可维护性改善，那就是值得的。Java 不是完美的，用 Java 一样可以写出不可维护的代码，但是 Java 至少比 Python 在这方面强很多。我们的经验证实了这点，信与不信就是看官的事了。

难得这么热烈的讨论，可是争论语言优劣没结果。

希望楼主再爆料点。这个大粉丝说的就是 Guido 了吧？这套系统说的就是 Rietveld？Java 的版本说的是 Gerrit 吗？可是 Gerrit 不是早就有了吗？难道你们内部用的不是开源的 Gerrit？另外，GAE 的 Python 版本你们内部有什么评价呢，比起 Java 和 Go 的呢？也觉得质量不行吗？Guido 干得不爽才走了吗？谢谢。

我用过 Rietveld，是 Python 语言本身在用的 review 工具，感觉风格挺老旧的。还没见有别的项目用这个东西。

Gerrit 是这个项目的 Java 的 fork，很多大项目在用。从这点说 Guido 开创的这个 Python 项目算成功的。当然后面的自然会吸取前面的教训，做得更好是应该的。

楼长说的那个系统应该就是 Mondrain,从来没有开源的.这里有详细的介绍,包括对 g 公司的 code review process.

http://www.youtube.com/watch?v=sMql3Di4Kgc
发信人: yueq (yueq), 信区: Python 标 题: Re: 终于把一个 8 万行的 Python 程序用 Java 重写了 发信站: 水木社区 (Sat Dec 7 16:09:34 2013), 转信

根据楼主的描述，是 Google 的内部 code review 工具：）。

发信人: yueq (yueq), 信区: Python 标 题: Re: 终于把一个 8 万行的 Python 程序用 Java 重写了 发信站: 水木社区 (Sat Dec 7 16:15:16 2013), 转信

旧系统 Mondrian 新系统 Critique

如大家上所述，我觉得楼主对于 UT、OOP 的理解还有待加强。

原文 http://blog.est.im/post/69161031446?utm_source=Tuicool_Weekly

Node.js 社区：一个人称代词引发的论战

作为一个开源项目，Node.js 以及相关项目都是由社区和志愿者共同维护的，任何的改动都会引发大家的讨论，尤其是现在 Node.js 在软件行业中用的越来越广，但是，5 天前的一个 Pull Request 请求被拒绝引发了轩然大波，整个事件的起因是有人希望在注释中使用“them”替换“him”，减少具有性别歧视的内容。

Rackspace 员工 Alex 首先提交了一个 pull request 请求，希望把 libuv（node.js 所依赖的库）注释中的“him”修改为“them”，大家对这样一个看似无关紧要的请求各持不同的意见，最终 libuv 的主要贡献者之一、StrongLoop 联合创始人 Ben Noordhuis 拒接了 Alex 的请求。
随后事件开始升级，大家讨论的中心转向了具有性别指向的名词对于女性权利的侵犯，对于请求被拒绝，Alex 表示十分困惑：

> 尽管这是一个微不足道的变化，但我还是很遗憾听到这个消息，我想不明白你为什么没有合并它。花费几秒钟时间按下一个“合并”按钮进而减少敌对的行为难道不是值得吗？

随着 Alex 支持者越来越多，Joyent 员工、Node.js 负责人 Isaac 开启了一个新的 Pull Request，确认从现有的代码中替换具有性别指向的人称代词。
由于 Node.js 是 Joyent 所创立并赞助的项目，任何相关的讨论都会最终引发对 Joyent 的口诛笔伐，因此 Joyent 的工程副总裁 Bryan Cantrill 及时针对这个事件写了一篇博客，表明了公司的立场：

> 如果 Ben 是我们的员工，我们会因此开除他……node.js 是一个开源项目，一个人没有那么大的力量。的确，一个依赖志愿者的开源项目所面临挑战之一就是处理这些棘手的问题，很庆幸 node.js 在 Isaac 妥善的掌握中。Isaac 是我所共事的工程师中最有包容性、善解人意的人之一，我知道他会本着 Node.js 的最佳利益妥善处理 Ben 的不妥当的行为。

同时 Ben 所在公司 StrongLoop 的 CEO Issac Roth 也及时出面发表了自己的看法，他说：

> 我从我的导师那里学到的一个事情是：如果有人犯了错误或者做了你认为不正确的事情，你可以私下和他们讨论，给他们机会改正它。
>
> ……
>
> Ben 没有理解人称代词变化的重要性，他正在尝试理解这些规则……Ben 为 libuv 贡献了 28%的代码，他是 Joyent 所赞助贡献者代码量总和的三倍……我认为他需要一个私下的讨论、一个电话或者一个学习的机会……我不能理解 Joyent 公开叫嚣开除别人公司雇员的做法……

两天前，Hacker News 上也引发了对这个问题的进一步讨论。
原文
http://www.infoq.com/cn/news/2013/12/the-power-of-a-pronoun?utm_source=Tuicool_Weekly

Github 项目中使用率最高的 Java/Ruby/JS 库

摘要：每个成功项目的背后都离不开一些库的支持，本文评选了 Github 上的三强语言 Java、Ruby、JavaScript 所开发的流行项目，并从这些流行项目里评选出最受欢迎的库。

提到开源，相信不少人会想到 Github，目前它的托管数量已经超过了一百万。上面几乎涵盖了各种类型的项目库，当然，参与贡献的开发者数量也是极多。因此，它也提供了相当多样的数据资源，一些研发人员利用这些数据来研究开/闭源、企业软件等发展趋势。

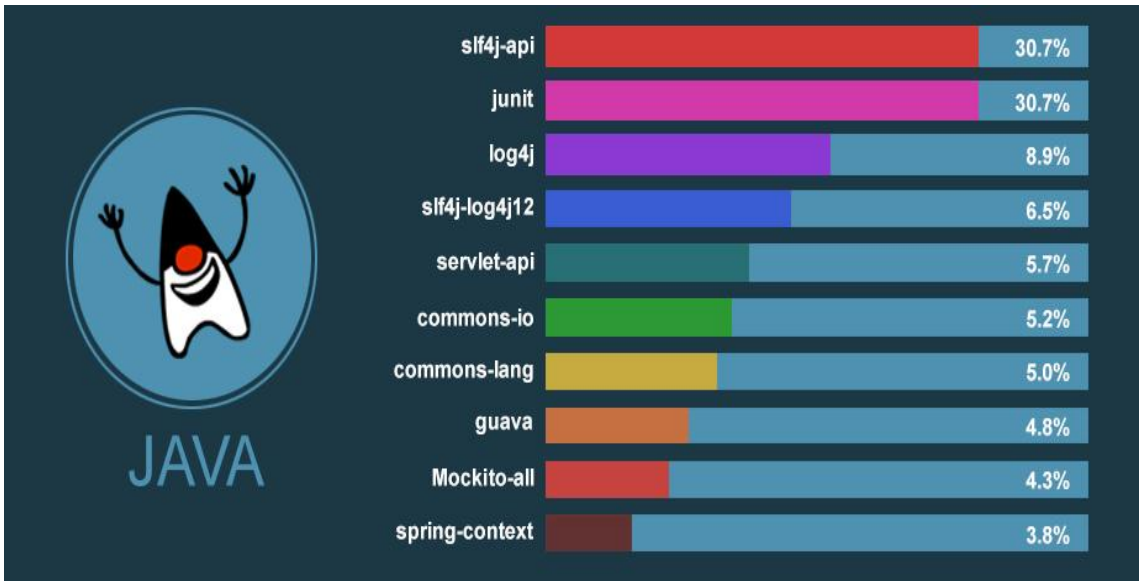
开发者每天都会面对一些软件开发库或开源库，在使用一些旧库的同时，每天都会出现许多新库，开发者如何在它们之间做出选择，找出适合自己的软件开发库呢？社区的用户数和贡献者的活跃度对一个库的成功起着非常关键的作用，这也是开发者选择的关键。对于一个开源库来说，开发者很容易知道它的贡献者数量，但一般很难知道有多少开发者使用它。本文作者采用具体的数据来解答这个问题。

作者选择了排名前三的编程语言库来进行分析，这三门语言分别是 Java、Ruby、JavaScript。分别分析了它们当中最流行的 10000 个项目（例如 Github 仓库），并且从中选出最受这些项目喜爱的库。此外，他们还分析了排名前 100 库所使用的组件、组合种类（例如测试、数据库、UI 等等），并且看看这些语言直接的不同之处。

下面分别列出了 10000 个 Java、Ruby、JavaScript 项目中使用率最高的 10 个库，并把一些值得特别关注的库和趋势进行总结和分析。

Java

注：点击链接可获得关于 Java 库的全部分析结果。



Java 项目中排名前十的库

Guava 是 Google 的开源库，目前，Google 代码已成为主流，虽然 Spring 和 Apache 库非常普遍，在前 100 个项目当中，它们占到了高于 25%的平均比例。但有些惊讶的是谷歌的一些库，例如 GWT 和 Guava，Guava 成功的排在了第 7 位，在 Java 的 10000 个项目中，有 4.8%的项目使用了它。

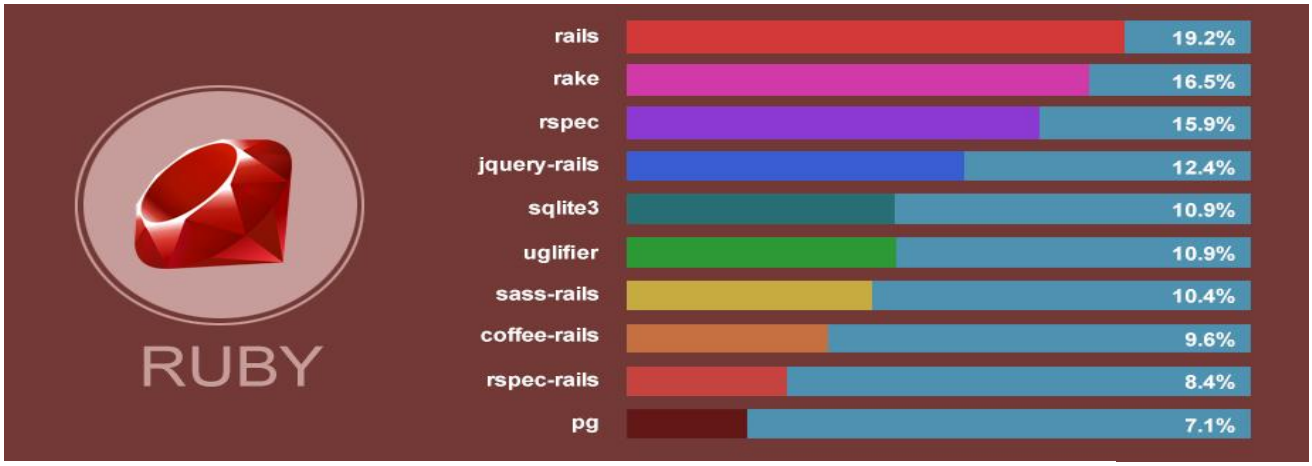
另外值得大家关注的一个库是 ElasticSearch，它是一款非常强大的搜索和数据分析引擎，目前在 Java 调查的项目里，有 110 个项目使用了该库。

数据处理占据了 Java 很大一部分，其中主要集中在数据管理。而在大数据方面，Hadoop 处于领先地位。在被调查的 10000 个项目中，有 168 个项目采用 Hadoop，最知名并且也是最常用的 SQL 数据库之一 MySql 被 225 项目采用，另一个知名的关系数据库 PostgreSQL 则是 121 个。

测试驱动开发在 Java 和 Ruby 中占据很大一部分，在这三门语言中，测试都扮演着非常重要的角色，在 Java 和 Ruby 中，40%-50%的项目都使用了自动测试框架进行项目复查工作。

Mocking 是一种在测试和开发中模拟真实世界对象的一种方法，目前该方法也得到了很多项目组的追捧，其中 Java 中有 10%、Ruby 里有 7%的项目在使用该方法。

Ruby



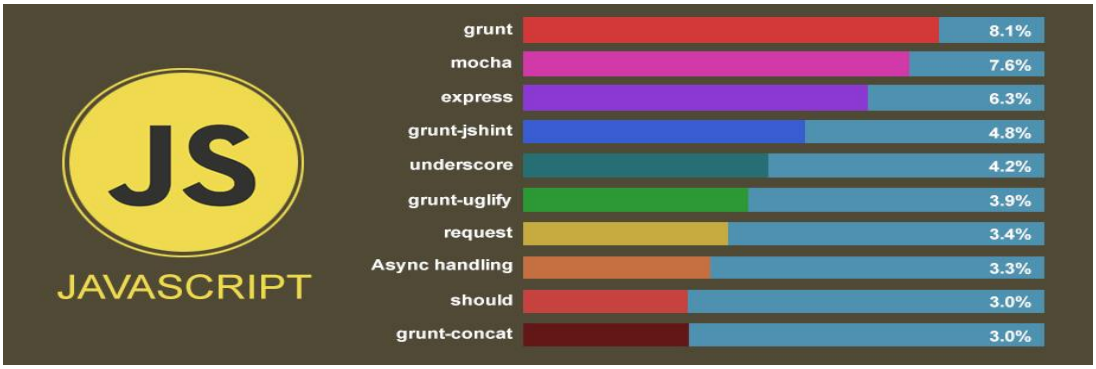
Ruby 项目里排名前十的库

在所调查的 10000 个 Ruby 项目中，虽然 NoSQL 数据库在这些天风靡一时，但关系数据库 SQL 在 Ruby 世界里仍在占主导地位——Sqlite、postgreSQL、在研究的项目中，有 25%的项目使用 MySql，而 Redis 和 MongoDB 仅占 3%，其中有 185 个项目采用 MongoDB 作为数据库，这个数据是 Java 项目里的两倍。

在 Web 开发方面，尽管越来越多的新框架在近几年得到追捧（例如有 570 个项目采用 Sinatra），但 Rails 仍然是 Ruby 的中心，有超过 7000 个项目使用它。Web 服务器方面，Thin（有 487 个项目）使用率则是 Unicorn 的 2 倍。CoffeeScript 似乎很受 Ruby 开发者的欢迎，拥有超过 1000 个项目使用。

Twitter 在 Ruby 中也有不小的影响力，在百强项目里，有 3 个库和 382 个项目使用它们。这是相当庞大的，但仍然没有谷歌在 Java 里的影响力大。

JavaScript



JavaScript 项目中排名前十的库

JavaScript 可以应用和支持更多类型的应用程序，但一些新特性和功能并未在语言和标准库上得到应用。因此，在研究中我们也看到，在 TOP100 项目里，有超过 50%的框架被 JavaScript 库所使用。

Grunt 自动化框架在 JS 开发中扮演着非常重要的角色（尤其是 node.js），在百强库当中，有 23%的库使用它。

在 JavaScript 库里面，有一大部分的库是用来进行网络和客户端/服务端通信的，数量是 Java 和 Ruby 里的 3 倍。这很有可能是因为 Web 开发人员不得不在浏览器端处理一些支离破碎的生态系统和相对较早的服务器堆栈。而对于服务器端 Web 开发，Node.js 的 express 框架占据领先地位，有 631 个项目采用它进行研发。

努力朝着结构化方向发展，JavaScript 在语言扩展上有很多优势，在调研的 1 万个项目中，有 844 个项目具有这一功能。此外，JavaScript 还是一门非常灵活的语言，开发者也正在使用更加结构化的方法来塑造它。Underscore.js 库提供了函数编程语言功能，类似于一些结构化的语言，例如 Scala，它在此次排行中处于第五位。

原文
http://www.csdn.net/article/2013-12-04/2817700-Top-10-Libraries-in-Java-JS-and-Ruby?utm_source=Tuicool_Weekly

Rails 3.2.16 / 4.0.2 发布，请尽快升级

Rails 团队今天发布了两个安全更新版本——Rails 3.2.16 和 4.0.2，修复了一些重要的安全漏洞。为了使升级更平滑，下面的链接中只提供了针对相应漏洞的补丁，你可以单独下载升级。

3.2.16、4.0.2 版本中修复的安全漏洞如下：

- CVE-2013-6417：查询生成器漏洞，影响全系版本。这个问题之前已经修复了（CVE-2013-0155），但未修复完整，一些第三方库可能会绕过保护。

- CVE-2013-4491：反射型 XSS 漏洞，存在于 Rails 国际化组件中，影响 3.0.6 及之后的版本。
- CVE-2013-6415：number_to_currency 助手中的 XSS 漏洞，影响全系版本。
- CVE-2013-6414：Action View 组件中的 DoS 漏洞，影响 3.0.0 及之后的版本。

除此之外，4.0.2 版本中还修复了如下安全漏洞：

- CVE-2013-6416：simple_format 助手中的 XSS 漏洞，影响 4.0.0 和 4.0.1 版本。

建议受影响的版本用户尽快升级，可**点击上面的链接，在打开的页面中下载相应的补丁。**

详细信息：Rails 3.2.16 / 4.0.2 release note
原文 http://www.iteye.com/news/28533?utm_source=Tuicool_Weekly

高效 jQuery 的奥秘

讨论 jQuery 和 javascript 性能的文章并不罕见。然而，本文我计划总结一些速度方面的技巧和我本人的一些建议，来提升你的 jQuery 和 javascript 代码。好的代码会带来速度的提升。快速渲染和响应意味着更好的用户体验。

首先，在脑子里牢牢记住 jQuery 就是 javascript。这意味着我们应该采取相同的编码惯例,风格指南和最佳实践。

首先，如果你是一个 javascript 新手，我建议您阅读 《JavaScript 初学者的最佳实践》,这是一篇高质量的 javascript 教程,接触 jQuery 之前最好先阅读。

当你准备使用 jQuery,我强烈建议你遵循下面这些指南:

缓存变量

DOM 遍历是昂贵的，所以尽量将会重用的元素缓存。

```
// 糟糕

h = $('#element').height();
$('#element').css('height',h-20);

// 建议
```

```
$element = $('#element');
h = $element.height();
$element.css('height',h-20);
```

避免全局变量

jQuery 与 javascript 一样，一般来说,最好确保你的变量在函数作用域内。

```
// 糟糕

$element = $('#element');
h = $element.height();
$element.css('height',h-20);

// 建议

var $element = $('#element');
var h = $element.height();
$element.css('height',h-20);
```

使用匈牙利命名法

在变量前加$前缀，便于识别出 jQuery 对象。

```
// 糟糕

var first = $('#first');
var second = $('#second');
var value = $first.val();

// 建议 - 在 jQuery 对象前加$前缀

var $first = $('#first');
var $second = $('#second'),
var value = $first.val();
```

使用 Var 链（单 Var 模式）

将多条 var 语句合并为一条语句，我建议将未赋值的变量放到后面。

```
var
```

```
  $first = $('#first'),
  $second = $('#second'),
  value = $first.val(),
  k = 3,
  cookiestring = 'SOMECOOKIESPLEASE',
  i,
  j,
  myArray = {};
```

请使用'On'

在新版 jQuery 中，更短的 on("click") 用来取代类似 click() 这样的函数。在之前的版本中 on() 就是 bind()。自从 jQuery 1.7 版本后，on() 附加事件处理程序的首选方法。然而，出于一致性考虑，你可以简单的全部使用 on()方法。

```
// 糟糕

$first.click(function(){
    $first.css('border','1px solid red');
    $first.css('color','blue');
});

$first.hover(function(){
    $first.css('border','1px solid red');
})

// 建议
$first.on('click',function(){
    $first.css('border','1px solid red');
    $first.css('color','blue');
})

$first.on('hover',function(){
    $first.css('border','1px solid red');
})
```

精简 javascript

一般来说,最好尽可能合并函数。

```
// 糟糕
```

```
$first.click(function(){
    $first.css('border','1px solid red');
    $first.css('color','blue');
});

// 建议

$first.on('click',function(){
    $first.css({
        'border':'1px solid red',
        'color':'blue'
    });
});
```

链式操作

jQuery 实现方法的链式操作是非常容易的。下面利用这一点。

```
// 糟糕

$second.html(value);
$second.on('click',function(){
    alert('hello everybody');
});
$second.fadeIn('slow');
$second.animate({height:'120px'},500);

// 建议

$second.html(value);
$second.on('click',function(){
    alert('hello everybody');
}).fadeIn('slow').animate({height:'120px'},500);
```

维持代码的可读性

伴随着精简代码和使用链式的同时，可能带来代码的难以阅读。添加缩紧和换行能起到很好的效果。

```
// 糟糕

$second.html(value);
$second.on('click',function(){
```

```
    alert('hello everybody');
}).fadeIn('slow').animate({height:'120px'},500);

// 建议

$second.html(value);
$second
    .on('click',function(){ alert('hello everybody');})
    .fadeIn('slow')
    .animate({height:'120px'},500);
```

选择短路求值

短路求值是一个从左到右求值的表达式，用 &&（逻辑与）或 || （逻辑或）操作符。

```
// 糟糕

function initVar($myVar) {
    if(!$myVar) {
        $myVar = $('#selector');
    }
}

// 建议

function initVar($myVar) {
    $myVar = $myVar || $('#selector');
}
```

选择捷径

精简代码的其中一种方式是利用编码捷径。

```
// 糟糕

if(collection.length > 0){..}

// 建议

if(collection.length){..}
```

繁重的操作中分离元素

如果你打算对 DOM 元素做大量操作（连续设置多个属性或 css 样式），建议首先分离元素然后在添加。

```
// 糟糕

var
    $container = $("#container"),
    $containerLi = $("#container li"),
    $element = null;

$element = $containerLi.first();
//... 许多复杂的操作

// better

var
    $container = $("#container"),
    $containerLi = $container.find("li"),
    $element = null;

$element = $containerLi.first().detach();
//... 许多复杂的操作

$container.append($element);
```

熟记技巧

你可能对使用 jQuery 中的方法缺少经验,一定要查看的文档,可能会有一个更好或更快的方法来使用它。

```
// 糟糕

$('#id').data(key,value);

// 建议 (高效)

$.data('#id',key,value);
```

使用子查询缓存的父元素

正如前面所提到的，DOM 遍历是一项昂贵的操作。典型做法是缓存父元素并在选择子元素时重用这些缓存元素。

```
// 糟糕

var
    $container = $('#container'),
    $containerLi = $('#container li'),
    $containerLiSpan = $('#container li span');

// 建议 (高效)

var
    $container = $('#container '),
    $containerLi = $container.find('li'),
    $containerLiSpan= $containerLi.find('span');
```

避免通用选择符

将通用选择符放到后代选择符中，性能非常糟糕。

```
// 糟糕

$('.container > *');

// 建议

$('.container').children();
```

避免隐式通用选择符

通用选择符有时是隐式的，不容易发现。

```
// 糟糕

$('.someclass :radio');

// 建议

$('.someclass input:radio');
```

优化选择符

例如，Id 选择符应该是唯一的，所以没有必要添加额外的选择符。

```
// 糟糕

$('div#myid');
$('div#footer a.myLink');

// 建议
$('#myid');
$('#footer .myLink');
```

避免多个 ID 选择符

在此强调，ID 选择符应该是唯一的，不需要添加额外的选择符，更不需要多个后代 ID 选择符。

```
// 糟糕

$('#outer #inner');

// 建议

$('#inner');
```

坚持最新版本

新版本通常更好：更轻量级，更高效。显然，你需要考虑你要支持的代码的兼容性。例如，2.0 版本不支持 ie 6/7/8。

摒弃弃用方法

关注每个新版本的废弃方法是非常重要的并尽量避免使用这些方法。

```
// 糟糕 - live 已经废弃

$('#stuff').live('click', function() {
  console.log('hooray');
});

// 建议
```

```
$('#stuff').on('click', function() {
  console.log('hooray');
});
// 注：此处可能不当，应为 live 能实现实时绑定，delegate 或许更合适
```

利用 CDN

谷歌的 CND 能保证选择离用户最近的缓存并迅速响应。（使用谷歌 CDN 请自行搜索地址，此处地址以不能使用，推荐 jquery 官网提供的 CDN）。

必要时组合 jQuery 和 javascript 原生代码

如上所述，jQuery 就是 javascript，这意味着用 jQuery 能做的事情，同样可以用原生代码来做。原生代码（或 vanilla）的可读性和可维护性可能不如 jQuery，而且代码更长。但也意味着更高效（通常更接近底层代码可读性越差，性能越高，例如：汇编，当然需要更强大的人才可以）。牢记没有任何框架能比原生代码更小，更轻，更高效（注：测试链接已失效，可上网搜索测试代码）。

鉴于 vanilla 和 jQuery 之间的性能差异，我强烈建议吸收两人的精华，使用（可能的话）和 jQuery 等价的原生代码。

最后忠告

最后，我记录这篇文章的目的是提高 jQuery 的性能和其他一些好的建议。如果你想深入的研究对这个话题你会发现很多乐趣。记住，jQuery 并非不可或缺，仅是一种选择。思考为什么要使用它。DOM 操作？ajax？模版？css 动画？还是选择符引擎？或许 javascript 微型框架或 jQuery 的定制版是更好的选择。

原文
http://www.cnblogs.com/yanhaijing/p/3458234.html?utm_source=Tuicool_Weekly

Jcrop 是一个功能强大的图像裁剪引擎

Jcrop 是一个功能强大的图像裁剪引擎 jQuery 的。

它的设计使开发人员可以很容易地直接集成了先进的图像裁剪功能集成到任何基于 Web 的应用程序在不牺牲动力和灵活性(或编码,测试和调试的星期)。Jcrop 还设有干净，组织良好的代码，在大多数现代的 web 浏览器效果很好。

在<HEAD>你需要加载必要文件的页面 这包括：

jQuery 库

Jcrop 的 Javascript

Jcrop CSS 样式表

它应该是这个样子：

[html] view plaincopyprint?

```
1. <script src="js/jquery.min.js"> </ SCRIPT>
2. <script src="js/jquery.Jcrop.min.js"> </ SCRIPT>
3. <link rel="stylesheet" href="css/jquery.Jcrop.css" type="tex
   t/css" />
```

入门第一个简单点的 Demo:

[java] view plaincopyprint?

```
1. <%@ page language="java" contentType="text/html; charset=UT
   F-8"
2.          pageEncoding="UTF-8"%>
3. <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional
   //EN" "http://www.w3.org/TR/html4/loose.dtd">
4. <html>
5. <head>
6. <meta http-equiv="Content-Type" content="text/html; charset=
   ISO-8859-1">
7. <title>Insert title here</title>
8.
9. <script src="http://code.jquery.com/jquery-1.10.1.min.js"></sc
   ript>
10.<script src="http://code.jquery.com/jquery-migrate-1.2.1.min.j
   s"></script>
11.<script src="js/jquery.Jcrop.min.js"></script>
12.<link rel="stylesheet" href="css/jquery.Jcrop.css" type="tex
   t/css" />
13.<script>
14.         jQuery(function(){
15.                         jQuery('#user_preview_img').Jcrop({
16.                                 trackDocument: true
17.                         });
18.                 });
19.</script>
20.</head>
21.<body>
22.         <img alt="" src="images/area1/1.jpg" id="user_previe
   w_img">
23.</body>
24.</html>
```

效果图:

jcrop 简单的事件处理 Demo:

[java] view plaincopyprint?

```
1. <%@ page language="java" contentType="text/html; charset=UTF-8"
2.          pageEncoding="UTF-8"%>
3. <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
4. <html>
5. <head>
6. <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
7. <title>Insert title here</title>
8. 
9. <script src="http://code.jquery.com/jquery-1.10.1.min.js"></script>
10. <script src="http://code.jquery.com/jquery-migrate-1.2.1.min.js"></script>
11. <script src="js/jquery.Jcrop.min.js"></script>
12. <link rel="stylesheet" href="css/jquery.Jcrop.css" type="text/css" />
13. <script>
14. jQuery(document).ready(function(){
15. 
16.          jQuery('#user_preview_img').Jcrop({
```

```
                onChange:  showCoords,
                onSelect:  showCoords
        });

});
//  Simple  event  handler,  called  from  onChange  and  onSelect
//  event  handlers,  as  per  the  Jcrop  invocation  above

function  showCoords(c)
{
        jQuery('#x1').val(c.x);
        jQuery('#y1').val(c.y);
        jQuery('#x2').val(c.x2);
        jQuery('#y2').val(c.y2);
        jQuery('#w').val(c.w);
        jQuery('#h').val(c.h);
};
</script>
</head>
<body>
      <div><img  alt=""  src="images/area1/1.jpg"  id="user_preview_img"></div>

                <form  onsubmit="return  false;"  class="coords">
                          <label>X1  <input  type="text"  size="4"  id="x1"  name="x1"  /></label>
                          <label>Y1  <input  type="text"  size="4"  id="y1"  name="y1"  /></label>
                          <label>X2  <input  type="text"  size="4"  id="x2"  name="x2"  /></label>
                          <label>Y2  <input  type="text"  size="4"  id="y2"  name="y2"  /></label>
                          <label>W  <input  type="text"  size="4"  id="w"  name="w"  /></label>
                          <label>H  <input  type="text"  size="4"  id="h"  name="h"  /></label>
                </form>

</body>
</html>
```

效果图：

jcrop 实例演示 Demo3：

[java] view plaincopyprint?

```
1. <%@ page language="java" contentType="text/html; charset=UTF-8"
2.          pageEncoding="UTF-8"%>
3. <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional //EN" "http://www.w3.org/TR/html4/loose.dtd">
4. <html>
5. <head>
6. <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
7. <title>Insert title here</title>
8. 
9. <script src="http://code.jquery.com/jquery-1.10.1.min.js"></script>
10. <script src="http://code.jquery.com/jquery-migrate-1.2.1.min.js"></script>
11. <script src="js/jquery.Jcrop.min.js"></script>
12. <link rel="stylesheet" href="css/jquery.Jcrop.css" type="text/css" />
13. <script>
14. jQuery(document).ready(function(){
```

```
15.
16.         jQuery('#user_preview_img').Jcrop({
17.                         onChange:  showCoords,
18.                         onSelect:  showCoords,
19.                         bgColor:  'red',
20.                         bgOpacity:  .4,
21.                         setSelect:  [  100,  100,  50,  50  ],
22.                         aspectRatio:  16  /  9
23.         });
24.
25.});
26.
27.//  Simple  event  handler,  called  from  onChange  and  onSel
   ect
28.//  event  handlers,  as  per  the  Jcrop  invocation  above

29.function  showCoords(c)
30.{
31.         jQuery('#x1').val(c.x);
32.         jQuery('#y1').val(c.y);
33.         jQuery('#x2').val(c.x2);
34.         jQuery('#y2').val(c.y2);
35.         jQuery('#w').val(c.w);
36.         jQuery('#h').val(c.h);
37.};
38.</script>
39.</head>
40.<body>
41.       <div><img  alt=""  src="images/area1/1.jpg"  id="user_pre
   view_img"></div>
42.
43.                 <form  onsubmit="return  false;"  class="coords
   ">
44.                           <label>X1  <input  type="text"  size="4
   "  id="x1"  name="x1"  /></label>
45.                           <label>Y1  <input  type="text"  size="4
   "  id="y1"  name="y1"  /></label>
46.                           <label>X2  <input  type="text"  size="4
   "  id="x2"  name="x2"  /></label>
47.                           <label>Y2  <input  type="text"  size="4
   "  id="y2"  name="y2"  /></label>
48.                           <label>W  <input  type="text"  size="4"
     id="w"  name="w"  /></label>
```

```
49.                         <label>H <input type="text" size="4" id="h" name="h" /></label>
50.                 </form>
51.
52.</body>
53.</html>
```

效果图：

Jcrop 实例 Demo4:

[java] view plaincopyprint?

```
1. <%@ page language="java" contentType="text/html; charset=UTF-8"
2.         pageEncoding="UTF-8"%>
3. <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional //EN" "http://www.w3.org/TR/html4/loose.dtd">
4. <html>
5. <head>
6. <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
7. <title>Insert title here</title>
8.
9. <script src="http://code.jquery.com/jquery-1.10.1.min.js"></script>
```

```
<script src="http://code.jquery.com/jquery-migrate-1.2.1.min.js"></script>
<script src="js/jquery.Jcrop.min.js"></script>
<link rel="stylesheet" href="css/jquery.Jcrop.css" type="text/css" />
<script type="text/javascript">
    jQuery(function($){

        // Create variables (in this scope) to hold the API and image size
        var jcrop_api,
                boundx,
                boundy,

                // Grab some information about the preview pane
                $preview = $('#preview-pane'),
                $pcnt = $('#preview-pane .preview-container'),
                $pimg = $('#preview-pane .preview-container img'),

                xsize = $pcnt.width(),
                ysize = $pcnt.height();

        console.log('init',[xsize,ysize]);
        $('#user_preview_img').Jcrop({
            onChange: updatePreview,
            onSelect: updatePreview,
            aspectRatio: xsize / ysize
        },function(){
            // Use the API to get the real image size
            var bounds = this.getBounds();
            boundx = bounds[0];
            boundy = bounds[1];
            // Store the API in the jcrop_api variable
            jcrop_api = this;

            // Move the preview into the jcrop container for css positioning
            $preview.appendTo(jcrop_api.ui.holder);
        });
```

```
45.
46.         function  updatePreview(c)
47.         {
48.             if  (parseInt(c.w)  >  0)
49.             {
50.                 var  rx  =  xsize  /  c.w;
51.                 var  ry  =  ysize  /  c.h;
52.
53.                 $pimg.css({
54.                     width:  Math.round(rx  *  boundx)  +  'px',
55.                     height:  Math.round(ry  *  boundy)  +  'px'
   ,
56.                     marginLeft:  '-'  +  Math.round(rx  *  c.x)
     +  'px',
57.                     marginTop:  '-'  +  Math.round(ry  *  c.y)
     +  'px'
58.                 });
59.             }
60.         };
61.
62.     });
63.
64.
65.</script>
66.
67.
68.<style  type="text/css">
69./*  Apply  these  styles  only  when  #preview-pane  has
70.       been  placed  within  the  Jcrop  widget  */
71..jcrop-holder  #preview-pane  {
72.     display:  block;
73.     position:  absolute;
74.     z-index:  2000;
75.     top:  10px;
76.     right:  -280px;
77.     padding:  6px;
78.     border:  1px  rgba(0,0,0,.4)  solid;
79.     background-color:  white;
80.
81.     -webkit-border-radius:  6px;
82.     -moz-border-radius:  6px;
83.     border-radius:  6px;
84.
```

```
    -webkit-box-shadow: 1px 1px 5px 2px rgba(0, 0, 0, 0.2);
    -moz-box-shadow: 1px 1px 5px 2px rgba(0, 0, 0, 0.2);
    box-shadow: 1px 1px 5px 2px rgba(0, 0, 0, 0.2);
}

/* The Javascript code will set the aspect ratio of the crop
    area based on the size of the thumbnail preview,
    specified here */
#preview-pane .preview-container {
    width: 250px;
    height: 170px;
    overflow: hidden;
}
</style>

</head>
<body>
        <div><img alt="" src="images/area1/1.jpg" id="user_preview_img"></div>

        <div id="preview-pane">
          <div class="preview-container">
              <img src="images/area1/1.jpg" class="jcrop-preview" alt="Preview" />
          </div>
        </div>
</body>
</html>
```

效果图:

注意：有关这些选项的对象的格式的几件事情：
文本值必须加引号（如'红'，'＃CCC“，”RGB（10,10,10）'）
数值，包括小数点，不应该被引用。
setSelect 带有一个数组，这里表示为一个数组文本
aspectRatio 可能是最简单的除以宽度/高度设置
后面没有逗号的最后一个选项

```
jQuery(function(){
});
```

全写为

```
jQuery(document).ready(function(){
});
```

原文
http://blog.csdn.net/kenhins/article/details/17136815?utm_source=Tuicool_Weekly

超载的 JavaScript 功能

一个 JavaScript 的优势在于它是弱类型，它允许很大的灵活性，不幸的是这意味着函数重载不可用。我们没有明确的参数类型声明，因此，我们不能“输入”参数时我们声明函数。
但是有多少，我们真的需要函数重载？我们不能没有它，我们可以简单地具备的功能检查传递的参数类型和参数长度以及基于这些，决定的逻辑。这实际上是有多少开发商选择做它，但是在我看来，这使得代码的可读性/维护，同时也增加了自定义代码，以每个需要重载的逻辑功能，这引入了层编码，可能引入错误。
因此，让我们试试，看看我们如何能够最好地在 JavaScript 中实现这一目标。
我环顾四周净，并有几个方法来实现这一目标。然而，他们没有那么灵活和快速实施，因为我想要的。如果我想从 A 点去 - > B 点在我的代码（有函数重载功能的 B 点），我想这样做，用最少的代码。我希望能够把它添加到我的已经存在的代码在尽可能短的时间内可能和我平时喜欢在这种事情在配置方式的约定。
在这篇文章中，我会要求你做一个函数调用你的 JavaScript 类，并与该呼叫时，您将自动拥有函数重载。仍然有你需要遵循一些约定，但它们是简单和直接的，有需要得到这个工作，几乎很少或根本没有配置。

我们需要什么？

首先，我们需要一种方法来通常写我们的代码没有超载，但应写入记住一些约定，允许超载可以很容易地添加。第二，我们需要一个简单的方式，从我们定义的参数“告诉”类型的信息。

对于第一步，我选择了写作，将被重载为一个函数不同功能的简单方法。在这里，我要编号的职能。假设你有一个名为 3 个不同功能的插件需要被重载，那么你就写你的原型功能的方式是通过增加一个后缀，以每个函数调用，这个后缀是一个简单的计数器。一个例子解释了这个最好的。我想有一个调用函数的 3 种不同口味加，那么我需要做的是创建一个名为三种功能 ADD1，ADD2 和 ADD3。后来，我的代码会去寻找这些“编号的函数”，并结合成一个重载的函数，它的名字是没有任何号码，以便：ADD1，ADD2，ADD3 … ADDX - > 添加（超载，为所有这些功能） 。

对于第二步，我需要一种简单，快速实施和简单的记住告诉每个参数的类型的方法。我的参数将开始与一个或多个字母，将描述的类型，然后停在第一个大写字母，这其中的参数的名称开始。举几个例子：nNumericParam，strName 中，oSomeObject 等等…在前面的例子中，这些参数类型“ Ñ “，” 海峡 “和” Ö “。我们将这些映射到实际类型的一个非常简单的对象，我们将传递到创建重载为我们的功能。

因此，让我们回顾一下，给我们如何需要编写一个类的小完整的例子。

```
// 一个无用的类真的，但一个很好的例子
function AdditionClass() {
  this.IsAdditionClass = true;
}
AdditionClass.prototype = {
  add1: function(nValue1, nValue2) {
    if (console)
      console.log('calling add1 with params:', arguments);
    return nValue1 + nValue2;
  },
  add2: function(nValue, strValue) {
    if (console)
      console.log('calling add2 with params:', arguments);
```

```
    return nValue + strValue;
  },
  add3: function(nValue1, nValue2, nValue3) {
    if (console)
      console.log('calling add3 with params:', arguments);
    return nValue1 + nValue2 + nValue3;
  }
}
```

在这个类中,我们写了这些功能 ADD1,ADD2 和 ADD3 将被过载到调用函数加载(请注意，你不应该创建或使用该函数名称添加在您的类，或者 overloader 将失败的功能）。我们还使用了匈牙利命名法，这将使我们能够“告诉”它的类型的命名我们的参数。

使用代码

要添加函数重载，我们需要调用只是一个功能。overloadPrototype。
这个函数有两个参数，第一个参数是类的构造函数，第二个参数是该匈牙利表示法前缀映射到实际的类型，你可以从得到一个对象 typeof 运算。您可能需要自定义类型匹配逻辑在自己的代码,所以为了方便您的实际匹配情况发生在一个叫做函数 checkTypesMatch 它获取传递的参数本身，它应该属于参数的匈牙利前缀。我想在此指出的参数检查并不总是采用，如果不同的功能超载具有独特的参数的长度，则该重载的代码将完全根据参数的长度决定，这将在后面详细在这篇文章中解释。
在前面的例子中，调用加上超载看起来像这样：

```
overloadFunction(AdditionClass, { n : "number", str : "string", o : "object" });
```

引擎盖下

那么，该函数 overloadFunction 真的。首先，它遍历传入构造函数的原型的功能，一旦它找到一个函数的最后一个字符为“ 1 “，它通过对这个函数名来命名函数 processFunctions，将尝试挂钩的过载。

该功能 processFunctions 开始获取传递。在这个例子中，函数名的基本名称，传递函数名“ ADD1 “是指基本名称是增加和新的重载函数将使用这个名称来创建。

第二，processFunctions 将循环并试图找到所有的计数功能，在本例中 ADD1，ADD2 和 ADD3。它将停止时，它无法找到任何更多的功能，或当计数值最大为 10 个功能（10 重载）。（需要注意的是维护你的代码的时候，你不能离开的差距在编号的功能）。我认为你不应该有超过 10 多载，因为这可能会使事情慢的重载功能，我们将在本文后面走了过来性能。如果你有超过 10 重载，你真的需要它们，你可能需要增加循环的上端。

第三，processFunctions 会整理此基础上他们的论据长度的函数列表，在这个过程中，它会检查，看看是否有重复的参数长度与否。在这里，我们有两种情况，重复的长度或没有，这是表示与变量 duplicateLen。

让我们以轻松的情况下第一，当我们没有重复的长度。这种情况下，更容易编写重载的函数和过载的逻辑也非常快，它的成本基本上是间接的函数调用只是另一个层次。processFunctions 现在交出构造函数，的计算功能和基本名称的数组要叫另一个函数 createLengthOverload。

该函数 createLengthOverload 创建一个具有基本名称（这里是“一个新的功能添加 “），这功能很简单，它会查找传递参数的长度并根据该会调用相应的原始功能。（这里是一个 ADD1，ADD2 和 ADD3）。

在我们的第二种情况，在那里我们有有一个共同的长度的功能，很明显，我们需要决定哪些功能基于类型的信息来调用，这就是事情变得丑陋，也只是一点点。 

在这种情况下，首先我们需要创建一个新的数据结构（称为数据中的代码），它整合了长度，这意味着我们组具有相同长度的功能。的长度 X 作为一个键和值的函数的信息的数组。

现在，我们需要去在这个新的数据结构（数据），并检查其长度有一个以上的功能，在我们上面的例子，ADD1 和 ADD2 都有参数长度 2 和 ADD3 具有参数长度 3。所以 ADD3 是安全的，它并没有对其参数的长度是竞争的其他功能，但是 ADD1 和 ADD2 做竞争，所以要解决哪一个打电话的方式将根据类型匹配的决定（次）。

对于每个组都有一个共同的长度的功能，我们现在创建一个新的数据结构，它们被称为 annotationArray，此架构将“描述”为每个函数参数的类型有（这里的两个函数 ADD1 和 ADD2）。该 annotationArray 将包含 ADD1：['号'，'数']和

ADD2：['号'，'字符串']。在这一点上，我们做了一些例行性检查，我们需要确保这些不同的功能至少有一个不同的参数类型，否则我们无法决定哪一个电话，至少一个参数应该是不同的每个不同的功能（在我们的例子中，第二个参数是数字 ADD1 和字符串 ADD2）。输入信息常用的参数下一个被删除（最优化标志设置），在我们的例子中，我们删除的类型信息的第一个参数（s）的 ADD1 和 ADD2，因为它们是两个数字，他们不为任何目的的决策，我们只是不停的第二个参数的信息。现在，我们已经准备好创建我们的重载函数，我们通过对数据结构以及基本名称和构造函数的功能 createTypeOverload。

该函数 createTypeOverload 将增加的基本名称（这里是“一个新的功能添加 “）相似，createLengthOverload 但逻辑是不同的。新创建的附加 功能，将有机会获得的（数据我们已经创建和早期抛光）的数据结构。在此基础上，并传递给它的参数的长度，这将决定要调用哪个函数（ADD1，ADD2 或 ADD3）。如果它是通过 3 个参数，它会直接调用 ADD3，如果通过了两项，它会经过候选函数，在这种情况下是 ADD1 和 ADD2，并会尝试消除那些不适合基于数据类型，最终只是一个函数应该赢得这场比赛。

性能

显然，事情不会发生神奇，有一个点球支付，问题是有多少是这个点球，是否愿意支付它。在其他语言中，编译器的重载让他们来免费的，在 JavaScript 中我们要做什么样的编译器，但大致在运行时。如果你有关键的代码，优化它没有函数重载将要走的路，当然，但是如果你的代码是不是很关键，让我们看看有多少是我们的惩罚。在这里的参数的长度是唯一在所有被重载的功能的情况下，这种情况下是速度快，非常快，几乎没有一个点球我会说。如果参数的长度都小于 10，我用一个数组的长度参数为指标，要调用的函数，所以这里的处罚仅仅是一个间接层是来自调用应用上的功能本身。如果参数的长度都大于 10，然后我用一个哈希得到的函数调用，这将是稍贵。当我们有共同的长度，我们必须去在不同的候选人，并消除它们一个接一个，所以根据不同的共享相同的参数长度函数的数目，刑罚会有所不同。如果我有时间，我会做一些压力测试，并发布一些数字对于一些常见的情况。

原文 http://ilixin.iteye.com/blog/1983972?utm_source=Tuicool_Weekly

Lua 简明教程

这几天系统地学习了一下 Lua 这个脚本语言，Lua 脚本是一个很轻量级的脚本，也是号称性能最高的脚本，用在很多需要性能的地方，比如：游戏脚本，nginx，wireshark 的脚本，当你把他的源码下下来编译后，你会发现解释器居然不到 200k，这是多么地变态啊（/bin/sh 都要 1M，MacOS 平台），而且能和 C 语言非常好的互动。我很好奇得浏览了一下 Lua 解释器的源码，这可能是我看过最干净的 C 的源码了。

我不想写一篇大而全的语言手册，一方面是因为已经有了（见本文后面的链接），重要的原因是，因为大篇幅的文章会挫败人的学习热情，我始终觉得好的文章读起来就像拉大便一样，能一口气很流畅地搞完，才会让人爽（这也是我为什么不想写书的原因）。所以，这必然又是一篇“入厕文章”，还是那句话，我希望本文能够让大家利用上下班，上厕所大便的时间学习一个技术。呵呵。

相信你现在已经在厕所里脱掉裤子露出屁股已经准备好大便了，那就让我们畅快地排泄吧……

运行

首先，我们需要知道，Lua 是类 C 的，所以，他是大小写字符敏感的。

下面是 Lua 的 Hello World。注意：Lua 脚本的语句的分号是可选的，这个和 GO 语言很类似。

```
1  print("Hello World")
```

你可以像 python 一样，在命令行上运行 lua 命令后进入 lua 的 shell 中执行语句。

```
1  chenhao-air:lua chenhao$ lua
2  Lua 5.2.2  Copyright (C) 1994-2013 Lua.org, PUC-Rio
3  > print("Hello, World")
4  Hello, World
5  >
```

也可以把脚本存成一个文件，用如下命令行来运行。

```
1  >lua   file.lua
```

或是像 shell 一样运行：

```
chenhao-air:lua chenhao$ cat hello.lua
#!/usr/local/bin/lua
print("Hello, World")
chenhao-air:lua chenhao$ chmod +x hello.lua
chenhao-air:test chenhao$ ./hello.lua
Hello, World
```

语法

注释

```
-- 两个减号是行注释
```

```
--[[
  这是块注释
  这是块注释
  --]]
```

变量

Lua 的数字只有 double 型，64bits，你不必担心 Lua 处理浮点数会慢（除非大于 100,000,000,000,000），或是会有精度问题。

你可以以如下的方式表示数字，0x 开头的 16 进制和 C 是很像的。

```
num = 1024
num = 3.0
num = 3.1416
num = 314.16e-2
num = 0.31416E1
num = 0xff
num = 0x56
```

字符串你可以用单引号，也可以用双引号，还支持 C 类型的转义，比如： ‘\a’（响铃）， ‘\b’ （退格）， ‘\f’ （表单）， ‘\n’ （换行）， ‘\r’（回车）， ‘\t’ （横向制表）， ‘\v’ （纵向制表）， ‘\\’ （反斜杠），‘\”‘ （双引号）， 以及 ‘\” （单引号）

下面的四种方式定义了完全相同的字符串(其中的两个中括号可以用于定义有换行的字符串)

```
1  a = 'alo\n123"'
2  a = "alo\n123\""
3  a = '\97lo\10\04923"'
4  a = [[alo
5  123"]]
```

C 语言中的 NULL 在 Lua 中是 nil,比如你访问一个没有声明过的变量,就是 nil,比如下面的 v 的值就是 nil

```
1  v = UndefinedVariable
```

布尔类型只有 nil 和 false 是 false,数字 0 啊, '' 空字符串('\0')都是 true!

另外,需要注意的是:lua 中的变量如果没有特殊说明,全是全局变量,那怕是语句块或是函数里。变量前加 local 关键字的是局部变量。

```
1  theGlobalVar = 50
2  local theLocalVar = "local variable"
```

控制语句

不多说了,直接看代码吧(注意:Lua 没有++或是+=这样的操作)

while 循环

```
1  sum = 0
2  num = 1
3  while num <= 100 do
4          sum = sum + num
5          num = num + 1
6  end
7  print("sum =",sum)
```

if-else 分支

```
1  if age == 40 and sex =="Male" then
2          print("男人四十一枝花")
```

```
3   elseif age > 60 and sex ~="Female" then
4           print("old man without country!")
5   elseif age < 20 then
6           io.write("too young, too naive!\n")
7   else
8           local age = io.read()
9           print("Your age is "..age)
10  end
```

上面的语句不但展示了 if-else 语句，也展示了
1）“～=”是不等于，而不是!=
2）io 库的分别从 stdin 和 stdout 读写的 read 和 write 函数
3）字符串的拼接操作符“..”

另外，条件表达式中的与或非为分是：and, or, not 关键字。

for 循环

从 1 加到 100

```
1   sum = 0
2   for i = 1, 100 do
3           sum = sum + i
4   end
```

从 1 到 100 的奇数和

```
1   sum = 0
2   for i = 1, 100, 2 do
3           sum = sum + i
4   end
```

从 100 到 1 的偶数和

```
1   sum = 0
```

```
2  for i = 100, 1, -2 do
3          sum = sum + i
4  end
```

until 循环

```
1  sum = 2
2  repeat
3        sum = sum ^ 2 --幂操作
4        print(sum)
5  until sum >1000
```

函数

Lua 的函数和 Javascript 的很像

递归

```
1  function fib(n)
2      if n < 2 then return 1 end
3      return fib(n - 2) + fib(n - 1)
4  end
```

闭包

同样，Javascript 附体！

示例一

```
1  function newCounter()
2          local i = 0
3          return function()         -- anonymous function
4                i = i + 1
5                  return i
6          end
7  end
8
9  c1 = newCounter()
10 print(c1())   --> 1
```

```
11  print(c1())   --> 2
```

示例二

```
1  function myPower(x)
2          return function(y) return y^x end
3  end
4
5  power2 = myPower(2)
6  power3 = myPower(3)
7
8  print(power2(4)) --4 的 2 次方
9  print(power3(5)) --5 的 3 次方
```

函数的返回值

和 Go 语言一样，可以一条语句上赋多个值，如：

```
1  name, age, bGay = "haoel", 37, false, "haoel@hotmail.com"
```

上面的代码中，因为只有 3 个变量，所以第四个值被丢弃。

函数也可以返回多个值：

```
1  function getUserInfo(id)
2          print(id)
3          return "haoel", 37, "haoel@hotmail.com", "http://coolshell.cn"
4  end
5
6  name, age, email, website, bGay = getUserInfo()
```

注意：上面的示例中，因为没有传 id，所以函数中的 id 输出为 nil，因为没有返回 bGay，所以 bGay 也是 nil。

局部函数

函数前面加上 local 就是局部函数，其实，Lua 中的函数和 Javascript 中的一个德行。

比如：下面的两个函数是一样的：

```
1  function foo(x) return x^2 end
2  foo = function(x) return x^2 end
```

Table

所谓 Table 其实就是一个 Key Value 的数据结构，它很像 Javascript 中的 Object，或是 PHP 中的数组，在别的语言里叫 Dict 或 Map，Table 长成这个样子：

```
1  haoel = {name="ChenHao", age=37, handsome=True}
```

下面是 table 的 CRUD 操作：

```
1  haoel.website="http://coolshell.cn/"
2  local age = haoel.age
3  haoel.handsome = false
4  haoel.name=nil
```

上面看上去像 C/C++中的结构体，但是 name,age, handsome, website 都是 key。
你还可以像下面这样写义 Table：

```
1  t = {[20]=100, ['name']="ChenHao", [3.14]="PI"}
```

这样就更像 Key Value 了。于是你可以这样访问：t[20]，t["name"], t[3.14]。

我们再来看看数组：

```
1  arr = {10,20,30,40,50}
```

这样看上去就像数组了。但其实其等价于：

```
1  arr = {[1]=10, [2]=20, [3]=30, [4]=40, [5]=50}
```

所以，你也可以定义成不同的类型的数组，比如：

```
1  arr = {"string", 100, "haoel", function() print("coolshell.cn") end}
```

注：其中的函数可以这样调用：arr[4]()。

我们可以看到 Lua 的下标不是从 0 开始的，是从 1 开始的。

```
1  for i=1, #arr do
2          print(arr[i])
3  end
```

注：上面的程序中：#arr 的意思就是 arr 的长度。

注：前面说过，Lua 中的变量，如果没有 local 关键字，全都是全局变量，Lua 也是用 Table 来管理全局变量的，Lua 把这些全局变量放在了一个叫“_G”的 Table 里。

我们可以用如下的方式来访问一个全局变量（假设我们这个全局变量名叫 globalVar）：

```
1  _G.globalVar
2  _G["globalVar"]
```

我们可以通过下面的方式来遍历一个 Table。

```
1  for k, v in pairs(t) do
2          print(k, v)
3  end
```

MetaTable 和 MetaMethod

MetaTable 和 MetaMethod 是 Lua 中的重要的语法，MetaTable 主要是用来做一些类似于 C++重载操作符式的功能。

比如，我们有两个分数：

```
1  fraction_a = {numerator=2, denominator=3}
2  fraction_b = {numerator=4, denominator=7}
```

我们想实现分数间的相加：2/3 + 4/7，我们如果要执行： fraction_a + fraction_b，会报错的。

所以，我们可以动用 MetaTable，如下所示：

```
1  fraction_op={}
2  function fraction_op.__add(f1, f2)
3          ret = {}
4          ret.numerator = f1.numerator * f2.denominator + f2.numerator * f1.denominat
5          ret.denominator = f1.denominator * f2.denominator
6          return ret
7  end
```

为之前定义的两个 table 设置 MetaTable：（其中的 setmetatble 是库函数）

```
1  setmetatable(fraction_a, fraction_op)
```

```
2  setmetatable(fraction_b, fraction_op)
```

于是你就可以这样干了：（调用的是 fraction_op.__add()函数）

```
1  fraction_s = fraction_a + fraction_b
```

至于__add 这是 MetaMethod，这是 Lua 内建约定的，其它的还有如下的 MetaMethod：

```
__add(a, b)                     对应表达式 a + b
__sub(a, b)                     对应表达式 a - b
__mul(a, b)                     对应表达式 a * b
__div(a, b)                     对应表达式 a / b
__mod(a, b)                     对应表达式 a % b
__pow(a, b)                     对应表达式 a ^ b
__unm(a)                        对应表达式 -a
__concat(a, b)                  对应表达式 a .. b
__len(a)                        对应表达式 #a
__eq(a, b)                      对应表达式 a == b
__lt(a, b)                      对应表达式 a < b
__le(a, b)                      对应表达式 a <= b
__index(a, b)                   对应表达式 a.b
__newindex(a, b, c)             对应表达式 a.b = c
__call(a, ...)                  对应表达式 a(...)
```

“面向对象”

上面我们看到有__index 这个重载，这个东西主要是重载了 find key 的操作。这操作可以让 Lua 变得有点面向对象的感觉，让其有点像 Javascript 的 prototype。（关于 Javascrip 的面向对象，你可以参看我之前写的 Javascript 的面向对象）

所谓__index，说得明确一点，如果我们有两个对象 a 和 b，我们想让 b 作为 a 的 prototype 只需要：

```
1  setmetatable(a, {__index = b})
```

例如下面的示例：你可以用一个 Window_Prototype 的模板加上__index 的 MetaMethod 来创建另一个实例：

```
1  Window_Prototype = {x=0, y=0, width=100, height=100}
```

```
2  MyWin = {title="Hello"}
3  setmetatable(MyWin, {__index = Window_Prototype})
```

于是：MyWin 中就可以访问 x, y, width, height 的东东了。（注：当表要索引一个值时如 table[key], Lua 会首先在 table 本身中查找 key 的值, 如果没有并且这个 table 存在一个带有__index 属性的 Metatable, 则 Lua 会按照__index 所定义的函数逻辑查找）

有了以上的基础，我们可以来说说所谓的 Lua 的面向对象。

```
1  Person={}
2
3  function Person:new(p)
4          local obj = p
5          if (obj == nil) then
6                  obj = {name="ChenHao", age=37, handsome=true}
7          end
8          self.__index = self
9          return setmetatable(obj, self)
10 end
11
12 function Person:toString()
13         return self.name .." : ".. self.age .." : ".. (self.handsome and "handsome
14 end
```

上面我们可以看到有一个 new 方法和一个 toString 的方法。其中：

1）self 就是 Person，Person:new(p)，相当于 Person.new(self, p)
2）new 方法的 self.__index = self 的意图是怕 self 被扩展后改写，所以，让其保持原样
3）setmetatable 这个函数返回的是第一个参数的值。

于是：我们可以这样调用：

```
1  me = Person:new()
2  print(me:toString())
3
```

```
4  kf = Person:new{name="King's fucking", age=70, handsome=false}
5  print(kf:toString())
```

继承如下，我就不多说了，Lua 和 Javascript 很相似，都是在 Prototype 的实例上改过来改过去的。

```
1  Student = Person:new()
2
3  function Student:new()
4          newObj = {year = 2013}
5          self.__index = self
6          return setmetatable(newObj, self)
7  end
8
9  function Student:toString()
10         return "Student : ".. self.year.." : " .. self.name
11 end
```

模块

我们可以直接使用 require(“model_name”)来载入别的 lua 文件，文件的后缀是.lua。载入的时候就直接执行那个文件了。比如：

我们有一个 hello.lua 的文件：

```
1  print("Hello, World!")
```

如果我们：require(“hello”)，那么就直接输出 Hello, World! 了。

注意：
1）require 函数，载入同样的 lua 文件时，只有第一次的时候会去执行，后面的相同的都不执行了。
2）如果你要让每一次文件都会执行的话，你可以使用 dofile(“hello”)函数
3）如果你要玩载入后不执行，等你需要的时候执行时，你可以使用 loadfile()函数，如下所示：

```
1  local hello = loadfile("hello")
2  ... ...
3  ... ...
```

```
4  hello()
```

loadfile(“hello”)后，文件并不执行，我们把文件赋给一个变量 hello，当 hello()时，才真的执行。（我们多希望 JavaScript 也有这样的功能（参看《Javascript 装载和执行》））

当然，更为标准的玩法如下所示。

假设我们有一个文件叫 mymod.lua，内容如下：

文件名：mymod.lua

```
1  local HaosModel = {}
2
3  local function getname()
4          return "Hao Chen"
5  end
6
7  function HaosModel.Greeting()
8          print("Hello, My name is "..getname())
9  end
10
11 return HaosModel
```

于是我们可以这样使用：

```
1  local hao_model = require("mymod")
2  hao_model.Greeting()
```

其实，require 干的事就如下：（所以你知道为什么我们的模块文件要写成那样了）

```
1  local hao_model = (function ()
2      --mymod.lua 文件的内容--
3  end)()
```

参考

我估计你差不多到擦屁股的时间了，所以，如果你还比较喜欢 Lua 的话，下面是几个在线文章你可以继续学习之：

- manual.luaer.cn lua 在线手册
- book.luaer.cn lua 在线 lua 学习教程
- lua 参考手册 Lua 参考手册的中文翻译（云风翻译版本）

关于 Lua 的标库，你可以看看官方文档：string， table， math， io， os。

原文
http://coolshell.cn/articles/10739.html?utm_source=Tuicool_Weekly

C/C++ Volatile 关键词深度剖析

背景
前几天，发了一条如下的微博（关于 C/C++ Volatile 关键词的使用建议）：

此微博，引发了朋友们的大量讨论：赞同者有之；批评者有之；当然，更多的朋友，是希望我能更详细的解读 C/C++ Volatile 关键词，来佐证我的微博观点。而这，正是我写这篇博文的初衷：本文，将详细分析 C/C++ Volatile 关键词的功能（有多种功能）、Volatile 关键词在多线程编程中存在的问题、Volatile 关键词与编译器/CPU 的关系、C/C++ Volatile 与 Java Volatile 的区别，以及 Volatile 关键词的起源，希望对大家更好的理解、使用 C/C++ Volatile，有所帮助。

Volatile，词典上的解释为：易失的；易变的；易挥发的。那么用这个关键词修饰的 C/C++变量，应该也能够体现出”易变”的特征。大部分人认识 Volatile，也是从这个特征出发，而这也是本文揭秘的 C/C++ Volatile 的第一个特征。

1. Volatile：易变的

在介绍 C/C++ Volatile 关键词的”易变”性前，先让我们看看以下的两个代码片段，以及他们对应的汇编指令（以下用例的汇编代码，均为 VS 2008 编译出来的 Release 版本）：

- **测试用例一：非 Volatile 变量**

代码	汇编

void main ()
{
int a = 5;
int b = 10;
int c = 20;
int d;
 	
scanf("%d", &c);
 	
a = fn(c);
	call dword ptr [__imp__scanf (12A20A8h)]
	mov eax,dword ptr [esp+8]
	add esp,8

b = a + 1;	lea ecx,[eax+1]

d = fn(b);
 	
cout << a << b << c << d;
}	

b = a + 1;这条语句，对应的汇编指令是：lea ecx, [eax + 1]。由于变量 a，在前一条语句 a = fn(c)执行时，被缓存在了寄存器 eax 中，因此 b = a + 1；语句，可以直接使用仍旧在寄存器 eax 中的 a，来进行计算，对应的也就是汇编：[eax + 1]。

- **测试用例二：Volatile 变量**

代码	汇编

void main ()
{
volatile inta = 5;
int b = 10;
int c = 20;
int d;
 	
scanf("%d", &c);	

a = fn(c);	mov ecx,dword ptr [esp+8] mov dword ptr [esp+0Ch],ecx
b = a + 1;	mov eax,dword ptr [esp+0Ch] ... inc eax

d = fn(b);
 	
cout << a << b << c << d;
}	

与测试用例一唯一的不同之处，是变量 a 被设置为 volatile 属性，一个小小的变化，带来的是汇编代码上很大的变化。a = fn(c)执行后，寄存器 ecx 中的 a，被写回内存：mov dword ptr [esp+0Ch], ecx。然后，在执行 b = a + 1；语句时，变量 a 有重新被从内存中读取出来：mov eax, dword ptr [esp + 0Ch]，而不再直接使用寄存器 ecx 中的内容。

1. **小结**

从以上的两个用例，就可以看出 C/C++ Volatile 关键词的第一个特性：易变性。所谓的易变性，在汇编层面反映出来，就是两条语句，下一条语句不会直接使用

上一条语句对应的 volatile 变量的寄存器内容，而是重新从内存中读取。
volatile 的这个特性，相信也是大部分朋友所了解的特性。

在了解了 C/C++ Volatile 关键词的”易变”特性之后，再让我们接着继续来剖析 Volatile 的下一个特性：”不可优化”特性。

1. Volatile：不可优化的

与前面介绍的”易变”性类似，关于 C/C++ Volatile 关键词的第二个特性：”不可优化”性，也通过两个对比的代码片段来说明：

- **测试用例三：非 Volatile 变量**

代码	汇编

void main ()
{
int a;
int b;
int c;
 	
a = 1;
b = 2;
c = 3;
 	
printf("%d, %d, %d", a, b, c);
	push 3
	push 2
	push 1
	call ...
	
	}

在这个用例中，非 volatile 变量 a，b，c 全部被编译器优化掉了（optimize out），因为编译器通过分析，发觉 a，b，c 三个变量是无用的，可以进行常量替换。最后的汇编代码相当简介，高效率。

- **测试用例四：Volatile 变量**

代码	汇编

void main ()
{
volatile int a;
volatile int b;
volatile int c;
 	
a = 1;
b = 2;
c = 3;
 	
printf("%d, %d, %d", a, b, c);
	mov eax,dword ptr [esp]
	mov ecx,dword ptr [esp+4]
	mov edx,dword ptr [esp+8]
	
	push eax
	push ecx
	push edx
	call ...
	
	}

测试用例四，与测试用例三类似，不同之处在于，a，b，c 三个变量，都是 volatile 变量。这个区别，反映到汇编语言中，就是三个变量仍旧存在，需要将三个变量从内存读入到寄存器之中，然后再调用 printf()函数。

1. 小结

从测试用例三、四，可以总结出 C/C++ Volatile 关键词的第二个特性：“不可优化”特性。volatile 告诉编译器，不要对我这个变量进行各种激进的优化，甚至将变量直接消除，保证程序员写在代码中的指令，一定会被执行。相对于前面提到的第一个特性：”易变”性，”不可优化”特性可能知晓的人会相对少一些。但是，相对于下面提到的 C/C++ Volatile 的第三个特性，无论是”易变”性，还是”不可优化”性，都是 Volatile 关键词非常流行的概念。

1. Volatile：顺序性

C/C++ Volatile 关键词前面提到的两个特性，让 Volatile 经常被解读为一个为多线程而生的关键词：一个全局变量，会被多线程同时访问/修改，那么线程内部，就不能假设此变量的不变性，并且基于此假设，来做一些程序设计。当然，这样的假设，本身并没有什么问题，多线程编程，并发访问/修改的全局变量，通常都会建议加上 Volatile 关键词修饰，来防止 C/C++编译器进行不必要的优化。但是，很多时候，C/C++ Volatile 关键词，在多线程环境下，会被赋予更多的功能，从而导致问题的出现。

回到本文背景部分我的那篇微博，我的这位朋友，正好犯了一个这样的问题。其对 C/C++ Volatile 关键词的使用，可以抽象为下面的伪代码：

```
代码

int something = 0;
volatile int flag = false;

Thread1 ()                              Thread2 ()
{                                       {
      // do something;                      if (flag == true)
      // 实际情况，肯定更加复杂             {
      something = 1;                            // assert something happens;
                                                // 实际情况，在假设something已经
      flag = true;                              // 发生的前提下，做接下来的工作
}                                               assert (something == 1);

                                                // do other things,depends on sth
                                                other things;
                                          }
                                    }
```

这段伪代码，声明另一个 Volatile 的 flag 变量。一个线程(Thread1)在完成一些操作后，会修改这个变量。而另外一个线程(Thread2)，则不断读取这个 flag 变量，由于 flag 变量被声明了 volatile 属性，因此编译器在编译时，并不会每次都从寄存器中读取此变量，同时也不会通过各种激进的优化，直接将 if (flag == true)改写为 if (false == true)。只要 flag 变量在 Thread1 中被修改，

Thread2 中就会读取到这个变化，进入 if 条件判断，然后进入 if 内部进行处理。在 if 条件的内部，由于 flag == true，那么假设 Thread1 中的 something 操作一定已经完成了，在基于这个假设的基础上，继续进行下面的 other things 操作。

通过将 flag 变量声明为 volatile 属性，很好的利用了本文前面提到的 C/C++ Volatile 的两个特性：”易变”性；”不可优化”性。按理说，这是一个对于 volatile 关键词的很好应用，而且看到这里的朋友，也可以去检查检查自己的代码，我相信肯定会有这样的使用存在。

但是，这个多线程下看似对于 C/C++ Volatile 关键词完美的应用，实际上却是有大问题的。问题的关键，就在于前面标红的文字：由于 flag = true，那么假设 Thread1 中的 something 操作一定已经完成了。flag == true，为什么能够推断出 Thread1 中的 something 一定完成了？其实既然我把这作为一个错误的用例，答案是一目了然的：这个推断不能成立，你不能假设看到 flag == true 后，flag = true;这条语句前面的 something 一定已经执行完成了。这就引出了 C/C++ Volatile 关键词的第三个特性：顺序性。

同样，为了说明 C/C++ Volatile 关键词的”顺序性”特征，下面给出三个简单的用例（注：与上面的测试用例不同，下面的三个用例，基于的是 Linux 系统，使用的是”GCC: (Debian 4.3.2-1.1) 4.3.2″）：

- **测试用例五：非 Volatile 变量**

代码	汇编

// cordering.c
 	
int A, B;
void foo()
{
A = B + 1;
B = 0;
}
	gcc -O2 -S -masm=intel cordering.c
	cat cordering.s
	
	mov eax, DWORD PTR B[rip]
	mov DWORD PTR B[rip], 0
	add eax, 1
	mov DWORD PTR A[rip], eax
	ret

一个简单的示例，全局变量 A，B 均为非 volatile 变量。通过 gcc O2 优化进行编译，你可以惊奇的发现，A，B 两个变量的赋值顺序被调换了！！！在对应的汇编代码中，B = 0 语句先被执行，然后才是 A = B + 1 语句被执行。

在这里，我先简单的介绍一下 C/C++编译器最基本优化原理：保证一段程序的输出，在优化前后无变化。将此原理应用到上面，可以发现，虽然 gcc 优化了 A，B 变量的赋值顺序，但是 foo()函数的执行结果，优化前后没有发生

任何变化，仍旧是 A = 1；B = 0。因此这么做是可行的。

- **测试用例六：一个 Volatile 变量**

代码	汇编
// cordering.c	gcc -O2 -S -masm=intel cordering.c

int A;
volatile int B;
void foo()
{
A = B + 1;
B = 0;
}
	mov eax, DWORD PTR B[rip]
	mov DWORD PTR B[rip], 0
	add eax, 1
	mov DWORD PTR A[rip], eax
	ret

此测试，相对于测试用例五，最大的区别在于，变量 B 被声明为 volatile 变量。通过查看对应的汇编代码，B 仍旧被提前到 A 之前赋值，Volatile 变量 B，并未阻止编译器优化的发生，编译后仍旧发生了乱序现象。

如此看来，C/C++ Volatile 变量，与非 Volatile 变量之间的操作，是可能被编译器交换顺序的。

通过此用例，已经能够很好的说明，本章节前面，通过 flag == true，来假设 something 一定完成是不成立的。在多线程下，如此使用 volatile，会产生很严重的问题。但是，这不是终点，请继续看下面的测试用例七。

- **测试用例七：两个 Volatile 变量**

代码	汇编
// cordering.c	gcc -O2 -S -masm=intel cordering.c

volatile int A;
volatile int B;
void foo()
{
A = B + 1;
B = 0;
}
	mov eax, DWORD PTR B[rip]
	add eax, 1
	mov DWORD PTR A[rip], eax
	mov DWORD PTR B[rip], 0
	ret

同时将 A，B 两个变量都声明为 volatile 变量，再来看看对应的汇编。奇迹发生了，A，B 赋值乱序的现象消失。此时的汇编代码，与用户代码顺序高度一直，先赋值变量 A，然后赋值变量 B。

如此看来，C/C++ Volatile 变量间的操作，是不会被编译器交换顺序的。

1. happens-before

通过测试用例六，可以总结出：C/C++ Volatile 变量与非 Volatile 变量间的操作顺序，有可能被编译器交换。因此，上面多线程操作的伪代码，在实际运行的过程中，就有可能变成下面的顺序：

```
代码

int something = 0;
volatile int flag = false;

Thread1 ()                                  Thread2 ()
{                                           {
      // do something;                              if (flag == true)
      // 实际情况，肯定更加复杂                     {
 顺                                                       // assert something happens;
 序[  flag = true;                                       // 实际情况，在假设something已经
 交[                                                     // 发生的前提下，做接下来的工作
 换   something = 1;                                     assert (something == 1);
}                                                        // do other things,depends on sth
                                                         other things;
                                            }
                                            }
```

由于 Thread1 中的代码执行顺序发生变化，flag = true 被提前到 something 之前进行，那么整个 Thread2 的假设全部失效。由于 something 未执行，但是 Thread2 进入了 if 代码段，整个多线程代码逻辑出现问题，导致多线程完全错误。

细心的读者看到这里，可能要提问，根据测试用例七，C/C++ Volatile 变量间，编译器是能够保证不交换顺序的，那么能不能将 something 中所有的变量全部设置为 volatile 呢？这样就阻止了编译器的乱序优化，从而也就保证了这个多线程程序的正确性。

针对此问题，很不幸，仍旧不行。将所有的变量都设置为 volatile，首先能够阻止编译器的乱序优化，这一点是可以肯定的。但是，别忘了，编译器编译出来的代码，最终是要通过 CPU 来执行的。目前，市场上有各种不同体系架构的 CPU 产品，CPU 本身为了提高代码运行的效率，也会对代码的执行顺序进行调整，这就是所谓的 CPU Memory Model (CPU 内存模型)。关于 CPU 的内存模型，可以参考这些资料：Memory Ordering From Wiki；Memory Barriers Are Like Source Control Operations From Jeff Preshing；CPU Cache and Memory Ordering From 何登成。下面，是截取自 Wiki 上的一幅图，列举了不同 CPU 架构，可能存在的指令乱序。

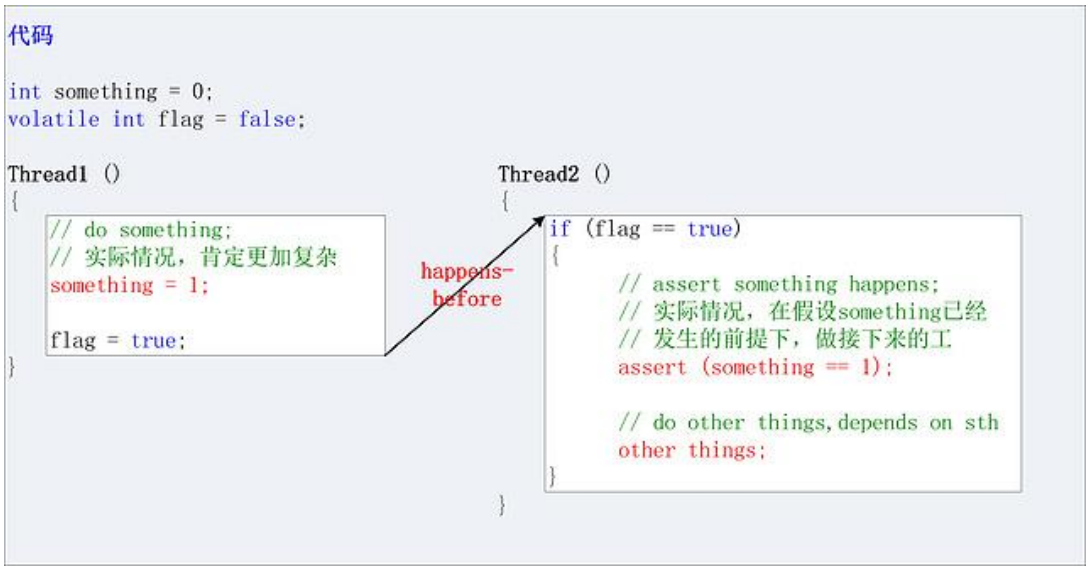
Memory ordering in some architectures[2][3]

Type	Alpha	ARMv7	PA-RISC	POWER	SPARC RMO	SPARC PSO	SPARC TSO	x86	x86 oostore	AMD64	IA-64	zSeries
Loads reordered after loads	Y	Y	Y	Y	Y				Y		Y	
Loads reordered after stores	Y	Y	Y	Y	Y				Y		Y	
Stores reordered after stores	Y	Y	Y	Y	Y	Y			Y		Y	
Stores reordered after loads	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Atomic reordered with loads	Y	Y		Y	Y						Y	
Atomic reordered with stores	Y	Y		Y	Y	Y					Y	
Dependent loads reordered	Y											
Incoherent Instruction cache pipeline	Y	Y		Y	Y	Y	Y	Y	Y		Y	Y

从图中可以看到，X86 体系(X86，AMD64)，也就是我们目前使用最广的 CPU，也会存在指令乱序执行的行为：StoreLoad 乱序，读操作可以提前到写操作之前进行。

因此，回到上面的例子，哪怕将所有的变量全部都声明为 volatile，哪怕杜绝了编译器的乱序优化，但是针对生成的汇编代码，CPU 有可能仍旧会乱序执行指令，导致程序依赖的逻辑出错，volatile 对此无能为力。

其实，针对这个多线程的应用，真正正确的做法，是构建一个 happens-before 语义。关于 happens-before 语义的定义，可参考文章：The Happens-Before Relation。下面，用图的形式，来展示 happens-before 语义：



如图所示，所谓的 happens-before 语义，就是保证 Thread1 代码块中的所有代码，一定在 Thread2 代码块的第一条代码之前完成。当然，构建这样的语义有很多方法，我们常用的 Mutex、Spinlock、RWLock，都能保证这个语义（关于 happens-before 语义的构建，以及为什么锁能保证 happens-before 语义，以后专门写一篇文章进行讨论）。但是，C/C++ Volatile 关键词不能保证这个语义，也就意味着 C/C++ Volatile 关键词，在多线程环境下，如果使用的不够细心，就会产生如同我这里提到的错误。

1. 小结

C/C++ Volatile 关键词的第三个特性：”顺序性”，能够保证 Volatile 变量间的顺序性，编译器不会进行乱序优化。Volatile 变量与非 Volatile 变量的顺序，编译器不保证顺序，可能会进行乱序优化。同时，C/C++ Volatile 关键词，并不能用于构建 happens-before 语义，因此在进行多线程程序设计时，要小心使用 volatile，不要掉入 volatile 变量的使用陷阱之中。

1. Volatile：Java 增强

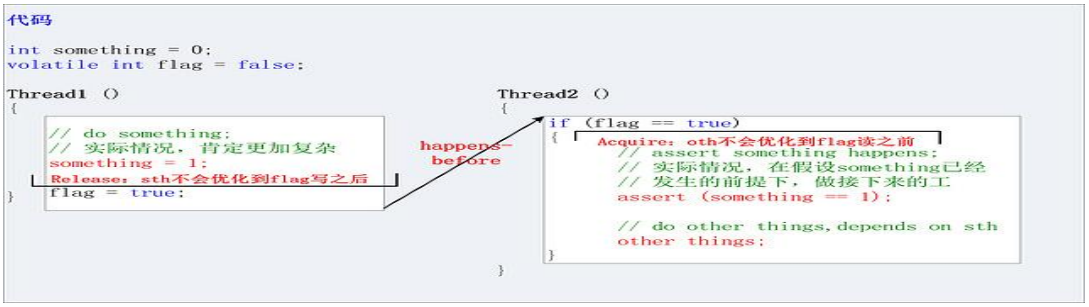
在介绍了 C/C++ Volatile 关键词之后，再简单介绍一下 Java 的 Volatile。与 C/C++的 Volatile 关键词类似，Java 的 Volatile 也有这三个特性，但最大的不同在于：第三个特性，”顺序性”，Java 的 Volatile 有很极大的增强，Java Volatile 变量的操作，附带了 Acquire 与 Release 语义。所谓的 Acquire 与 Release 语义，可参考文章：Acquire and Release Semantics。(这一点，后续有必要的话，可以写一篇文章专门讨论)。Java Volatile 所支持的 Acquire、Release 语义，如下：

- 对于 Java Volatile 变量的写操作，带有 Release 语义，所有 Volatile 变量写操作之前的针对其他任何变量的读写操作，都不会被编译器、CPU 优化后，乱序到 Volatile 变量的写操作之后执行。

- 对于 Java Volatile 变量的读操作，带有 Acquire 语义，所有 Volatile 变量读操作之后的针对其他任何变量的读写操作，都不会被编译器、CPU 优化后，乱序到 Volatile 变量的读操作之前进行。

通过 Java Volatile 的 Acquire、Release 语义，对比 C/C++ Volatile，可以看出，Java Volatile 对于编译器、CPU 的乱序优化，限制的更加严格了。Java Volatile 变量与非 Volatile 变量的一些乱序操作，也同样被禁止。

由于 Java Volatile 支持 Acquire、Release 语义，因此 Java Volatile，能够用来构建 happens-before 语义。也就是说，前面提到的 C/C++ Volatile 在多线程下错误的使用场景，在 Java 语言下，恰好就是正确的。如下图所示：



1. Volatile 的起源

C/C++的 Volatile 关键词，有三个特性：易变性；不可优化性；顺序性。那么，为什么 Volatile 被设计成这样呢？要回答这个问题，就需要从 Volatile 关键词的产生说起。(注：这一小节的内容，参考自 C++ and the Perils of Double-Checked Locking 论文的第 10 章节：volatile：A Brief History。这是一篇顶顶好的论文，值得多次阅读，强烈推荐！)

Volatile 关键词，最早出现于 19 世纪 70 年代，被用于处理 memory-mapeed I/O (MMIO)带来的问题。在引入 MMIO 之后，一块内存地址，既有可能是真正的内存，也有可能被映射到一个 I/O 端口。相对的，读写一个内存地址，既有可能操作内存，也有可能读写的是一个 I/O 设备。MMIO 为什么需要引入 Volatile 关键词？考虑如下的一个代码片段：

```
unsigned int *p = GetMagicAddress();
unsigned int a, b;

a = *p;                    (1)
b = *p;                    (2)

*p = a;                    (3)
*p = b;                    (4)
```

在此代码片段中，指针 p 既有可能指向一个内存地址，也有可能指向一个 I/O 设备。如果指针 p 指向的是 I/O 设备，那么(1)，(2)中的 a，b，就会接收到 I/O 设备的连续两个字节。但是，p 也有可能指向内存，此时，编译器的优化策略，就可能会判断出 a，b 同时从同一内存地址读取数据，在做完(1)之后，直接将 a 赋值给 b。对于 I/O 设备，需要防止编译器做这个优化，不能假设指针 b 指向的内容不变——易变性。

同样，代码(3)，(4)也有类似的问题，编译器发现将 a，b 同时赋值给指针 p 是无意义的，因此可能会优化代码(3)中的赋值操作，仅仅保留代码(4)。对于 I/O 设备，需要防止编译器将写操作给彻底优化消失了——”不可优化”性。

对于 I/O 设备，编译器不能随意交互指令的顺序，因为顺序一变，写入 I/O 设备的内容也就发生变化了——”顺序性”。

基于 MMIO 的这三个需求，设计出来的 C/C++ Volatile 关键词，所含有的特性，也就是本文前面分析的三个特性：易变性；不可优化性；顺序性。

1. 参考资料

[1] Wiki. Volatile variable.
[2] Wiki. Memory ordering.

[3] Scott Meyers; Andrei Alexandrescu. C++ and the Perils of Double-Checked Locking.
[4] Jeff Preshing. Memory Barriers Are Like Source Control Operations.
[5] Jeff Preshing. The Happens-Before Relation.
[6] Jeff Preshing. Acquire and Release Semantics.
[7] 何登成. CPU Cache and Memory Ordering——并发程序设计入门.
[8] Bartosz Milewski. Who ordered sequential consistency?
[9] Andrew Haley. What are we going to do about volatile?
[10] Java Glossary. volatile.
[11] stackoverflow. Why is volatile not considered useful in multithreaded C or C++ programming?
[12] msdn. Volatile fields.
[13] msdn. volatile (C++).
[14] 刘未鹏. 《C++0x 漫谈》系列之：多线程内存模型.
原文

http://hedengcheng.com/?p=725&utm_source=Tuicool_Weekly

使用 Key Collection 提高 Java 集合操作效率

Java Collections 框架中的类应该是整个 JDK 中使用频率最高的。然而，Java Collections 中的接口和实现存在着一些疏漏和弊端，大量的第三方库都致力于解决这些问题。
Brownies Collections 已经提供了一个轻量高效的 List 实现——GapList（详情见 java.dzone.com/articles/gaplist-lightning-fast-list）。GapList 是 ArrayList 和 LinkedList 的替代选择，旨在解决一些已知的性能问题。
由于 GapList 能提升应用的性能，这里引入的键集合（Key Collections）将会提升开发者的工作效率。为了实现这一目标，这些集合中集成了键和约束的概念，并能以一种正交和声明式的方式使用。

听上去很有趣，但是不是太理论化了？我们来看一个例子。

第一个例子
比方说，我们需要用一个 List 按照下面的要求表示表格的列：

- 列是有序的
- 列名必须唯一
- 能够通过下标（Index）和名称快速访问列

这样的话，我们应该选择什么集合呢？一个支持下标快速访问的 List。如果你确定元素的数量一直很少，我们能够用遍历的方式实现根据名称访问元素（同时需要用遍历的方式检查名称的唯一性），但这明显需要实现一个不能扩展的解决

方案。对于一个可扩展的解决方案，我们需要同步一个 List 和一个 Map。这并不是一个不能完成的任务，但是通常情况下会花费很大的工作量。

输入我们的键集合。只用一条语句就能创建满足需求的集合：

```
Key1List<Column,String>          columns          =          new
Key1List.Builder<Column,String>.
withKey1Map(Column.Mapper).withKey1Null(false).
withKey1Duplicates(false).build()
```

这段代码简单易懂：创建一个用于存储列对象的 List，允许通过指定 Mapper 所定义的字符串类型的键访问元素。键不能为空且不能重复。
要了解完整的示例代码，请看下面的 Column 类和 Mapper 的定义：

```


class Column {
        static     Mapper<Column,String>     Mapper     =     new
Mapper<Column,String>() {
                String getKey(Column col) {
                        return col.getName();
                }
        };

        private String name;

        public String getName() { return name; }
}


```

起初，你可能会认为这种集合创建的方式很笨拙，但如果不这么做，想想你需要耗费的代码量吧。此外，Java 8 中引入的 Lambda 表达式将能用方法引用替代 Mapper 的显式定义。

键

在看完第一个例子之后，我们来看看关于键更详细的定义和功能：键是从集合内的某个元素中提取出的一个值。它用于访问元素和定义约束。我们把集合内提取出的所有键值称为键映射（Key Map）。每个集合可以有一个或多个键映射，并且元素本身也能够作为键，我们把这种集合称作元素集（Element Set）。

对于每一个键映射，可以定义如下行为：

- 空值：允许插入空值（默认）或者通过 withKeyNull()方法禁止插入空值。
- 重复值：允许插入重复值（默认）或者通过 withKeyDuplicates ()方法禁止插入重复值。还存在一种模式，禁止插入重复值，但是空值可以重复插入。

- 排序顺序：键的顺序可以是随机的（默认）或是有序的。可以通过 withKeySort()方法实现键按照自然序或者其他自定义比较器产生的顺序有序。

如果你设定了一个键映射的排序顺序，你同样能够通过 withKeyOrderBy()方法指定整个集合的顺序。

如果你有一个后台数据库，这些概念对你来说听起来很熟悉。在我们的例子中，我们定义了列名作为集合的主键。如果想要让主键变得更明显或是更简短，我们可以使用 withPrimarKey()设定主键。还可以使用另一个方法 withUniqueKey()，withUniqueKey()定义的键可以为空，但非空的键值必须是唯一的。

需要注意的是键的取值应该是不可变的。这是因为键映射在元素添加时被填入键值，而键值后续的变更将不会被检测到。然而，这并不是键集合所特有的问题，在 Set 和 Map 中，元素的键值也是不能变更的。如果你真的需要变更一个键的值，你可以使用 invalidateKey()方法更新键集合的内部状态。

约束

对空值和重复值的限制已经展示出约束的力量。不幸的是，JDK 类库中并没有约束集合的概念。尽管有些类（如 Set）具有内部约束，但没有一种通用的方式去设定集合的约束。更重要的是，这些类的设计并不具备扩展性，因此实现一个约束会涉及到很多方法的重写。

也许你不知道为什么我们无论如何都需要带约束的集合，但对于优秀的 API 来说，这是一个至关重要的功能。我们设想这样一个场景，我们的类需要维护一个任务列表，任务可能不为空。客户端代码能够通过所提供的 API 对这些任务进行读写操作，但是必须保证只有非空任务才能被添加到任务列表中。

```
1
class Task {
2
        List<Task> tasks;
3
        List<Task> getTasks() { return tasks; }
}
4
```

上面的代码很明显是不安全的：客户端拥有不受限的列表写权限，这样她就能轻松地插入插入空值。我们可以用一个略有修改的版本来获得一个较好的只读接口：

```
1
List<Task> getTasks() { return Collections.unmodifiableList(tasks); }
```

但是我们还是需要完善 API 写操作的部分。这里有两种选择：

懒惰的开发者会只提供一套精简的 API:

```
1
void addTask(int index, Task task) {
2
        checkTask(task);
3
        tasks.add(index, task);
}
4
void setTask(int index, Task task) {
```

```
        checkTask(task);
        tasks.set(index, task);
}
```

勤劳的开发者会实现所有 Mutator 方法（两种 add()方法，两种 addAll()方法，set()方法）。或者她只提供一个方法。

```
void setTasks(List<Task> tasks) {
        checkTasks(tasks);
        this.tasks = tasks;
}
```

采用这种实现方式，我们必须在每次变更操作之后检查所有任务的合法性。在元素数量较少并且检查操作相对简单（如检查空值）的时候，这或许不是问题，但很明显这种实现方式不具备扩展性。

然而，当使用可约束的集合时，我们的 API 变得如此简单：

```
List<Task> getTasks() { return constraintTasks; }
```

因为列表本身可以检查元素的合法性。

因此，键集合提供了一个通用的概念队存储在集合中的元素建立约束。默认情况下，键集合就像 Collection 和 List 一样允许插入空值和重复值。与此同时，禁止插入空值(使用 withElemNull()方法)和重复值(使用 withElemDuplicates ()方法）也很容易。

你也可以使用 withConstraint()方法定义一个谓词约束，谓词约束用于判断一个新元素是否能加入集合。如果元素不满足谓词约束，该操作将以抛异常的形式被取消。

最后，还有一种约束可以使用。你可以通过定义集合的最大尺寸限制集合中元素的数量（使用 withMaxSize()方法）：试图加入更多元素的操作将以抛异常的形式被拒绝。如果你的集合是 List 类型，你还可以定义一个窗口尺寸（使用 withWindowSize ()方法）：窗口尺寸同样用于限制元素的最大数量，不同之处在于，新元素加入后，如果元素数量超出了窗口尺寸，集合的第一个元素将会被自动删除。

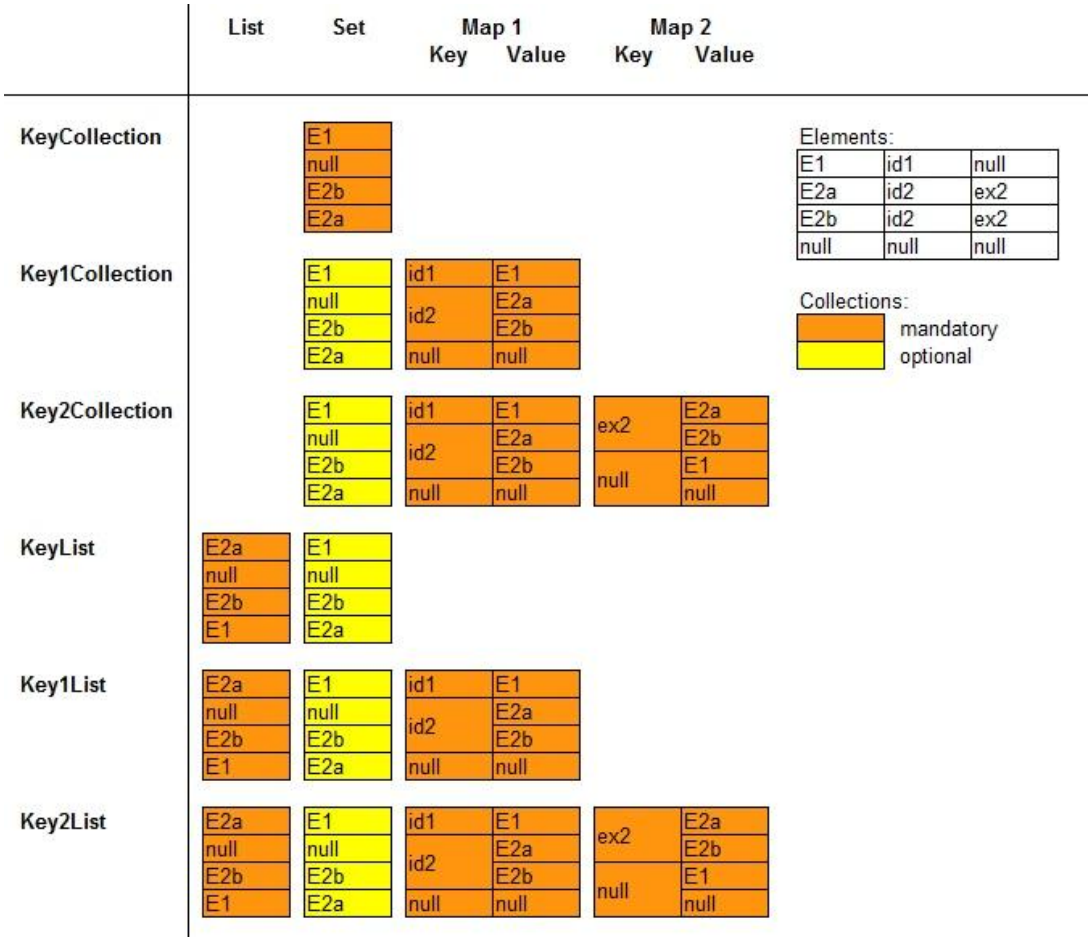
类

下表中的类提供键和约束的功能：

类	描述
KeyCollection<E>	维护一个元素集合，提供对元素的快速访问。
Key1Collection<E,K1>	带有一个附加键的键集合，提供对键的快速访问，也可以提供对元素本身的快速访问。
Key2Collection<E,K1,K2>	带有两个附加键的键集合，提供对键的快速访问，也可以提供对元素本身的快速访问。
KeyList<E>	维护一个元素列表，可以提供对元素本身的快速访问。
Key1List<E,K1>	带有一个附加键的键列表，提供对键的快速访问，也可以提供对元素本身的快速访问。
Key2List<E,K1,K2>	带有两个附加键的键列表，提供对键的快速访问，也可以提供对元素本身的快速访问。

键集合的代码实现中使用了 JDK 提供的集合类：使用 HashSet/TreeSet 实现元素集（Element Set），使用 HashMap/TreeMap 实现键映射（Key Map）。

下图说明了键集合如何运用不同的组件来存储带有两个键的元素：



KeyCollection，Key1Collection 和 Key2Collection 实现了 Collection 接口。默认情况下，元素的顺序由 Set 组件决定，但也可以按照键映射的顺序排列。KeyList，Key1List 和 Key2List 实现了 List 接口，因此元素的顺序始终由 List 决定。不过，列表中的元素也可以按照元素集或者键映射的顺序排列。
这些方法可以用于访问键映射中的键值： containsKey()、getByKey()、getAllByKey()、 getCountByKey()、getDistinctKeys()、removeByKey()、removeAllByKey()和 indexOfKey() （仅用于 List）。

这些方法可以用于访问元素集：getAll()、getCount()、removeAll()和 getDistinct()。

你也可以通过 asSet()和 asMap()方法获得 JDK 接口类型的对象来访问元素集和键映射中的值。

键列表

所有的非有序键列表用键值删除元素会比较慢，这是由于非有序键列表只能通过遍历的方式找到待删除元素。因此，你应该仅在元素数量较小并且只用下标删除元素时使用键列表。

有序键列表

有关“为什么 Java 语言没有有序列表”的讨论很多。反对的理由主要有两点：第一点出于设计理念的考虑，有序列表违背了 List.add()的设计理念，因为元素并没有从列表的尾端加入；第二点出于性能的考虑，以随机顺序向有序列表添加元素的性能开销会比较大。
尽管这两个理由都是对的，但有序键列表仍然会被使用，因为它方便使用。然而，你必须明白的是，有序键列表内部是基于一个有序 Map 实现的，以随机顺序向有序键列表添加元素的效率将会一直很低，因为元素添加的过程中必须移动内存中的其他元素。

更多的例子

最后，让我们用一些展现键集合力量的例子来对本文做个总结：

一个车票集合，每张票有一个强制的内部 ID 和一个可选的外部 ID：

```
1
2  Key2Collection<Ticket,String,String> coll =
   new Key2Collection.Builder<Ticket,String,String>.
3  withKey1Map(Ticket.dMapper).withPrimaryKey().
   withKey2Map(Ticket.ExtIdMapper).withUniqueKey2().build()
4
```

一个按列名排序的列表：

```
1  Key1List<Column,String> list = new Key1List<Column,String>.
2  withKey1Map(Column.Mapper).withKey1Sort(true).withKey1OrderBy(true).withPrimaryK
```

一个有序的原生类型（primitive）的整数列表：

需要注意的是，列表支持使用原生类型数值进行排序：

```
KeyList<Integer> list = new KeyList<Integer>.withElemOrderBy(int.class).build()
```

KeyList 的内部使用 IntObjGapList 存储列表元素。因此，存储 1,000,000 个元素只需要 4 Mbytes 内存空间，而标准的实现方式至少需要 44 Mbytes。

```
Set<Integer> set = new HashSet<Integer>()
```

但是，要想获得这样的性能收益，你需要在你的数据加入列表之前，对它们进行排序。

带约束的 List:

```
KeyList<String> = new KeyList<String>.withConstraint(uppercasePredicate).build()
```

带约束的 Set:

```
KeyCollection<String> = KeyCollection<String>.withConstraint(uppercasePredicate).build().asSet()
```

带约束的 Map:

```
KeyICollection<Column,String> = new KeyICollection<Column,String>.
withKeyIMap(nameMapper).withConstraint(uppercasePredicate).
build().asMapI()
```

现在，从 www.magicwerk.org/collections 下载最新版本的 Brownies Collections，享受它带来的效率提升。

原文 http://www.importnew.com/7528.html?utm_source=Tuicool_Weekly

在 iOS 中创建静态库

如果你作为 iOS 开发者已经有一段时间，可能会有一套属于自己的类和工具函数，它们在你的大多数项目中被重用。重用代码的最简单方法是简单的 拷贝/粘贴 源文件。然而，这种方法很快就会成
“”

如果你作为 iOS 开发者已经有一段时间，可能会有一套属于自己的类和工具函数，它们在你的大多数项目中被重用。

重用代码的最简单方法是简单的 拷贝/粘贴 源文件。然而，这种方法很快就会成为维护时的噩梦。因为每个 app 都有自己的一份代码副本,你很难在修复 bug 或者升级时保证所有副本的同步。

这就是静态库要拯救你的。一个静态库是若干个类,函数,定义和资源的包装，你可以将其打包并很容易的在项目之间共享。

在本教程中，你将用两种方法亲手创建你自己的通用静态库。

为了获得最佳效果，你应该熟悉 Objective-C 和 iOS 编程。Core Image 的相关知识并不是必须的，但是如果你对示例工程和滤镜代码如何工作感兴趣，了解它会有所帮助。

准备好以效率的名义减少，重用并再生你的代码!

为什么使用静态库

创建静态库可能出于以下几个理由:
1.你想将一些你和你团队中的同事们经常使用的类打包并轻松的分享给周围其他人。
2.你想让一些通用代码处于自己的掌控之下，以便于修复和升级。
3.你想将库共享给其他人，但不想让他们看到你的源代码。

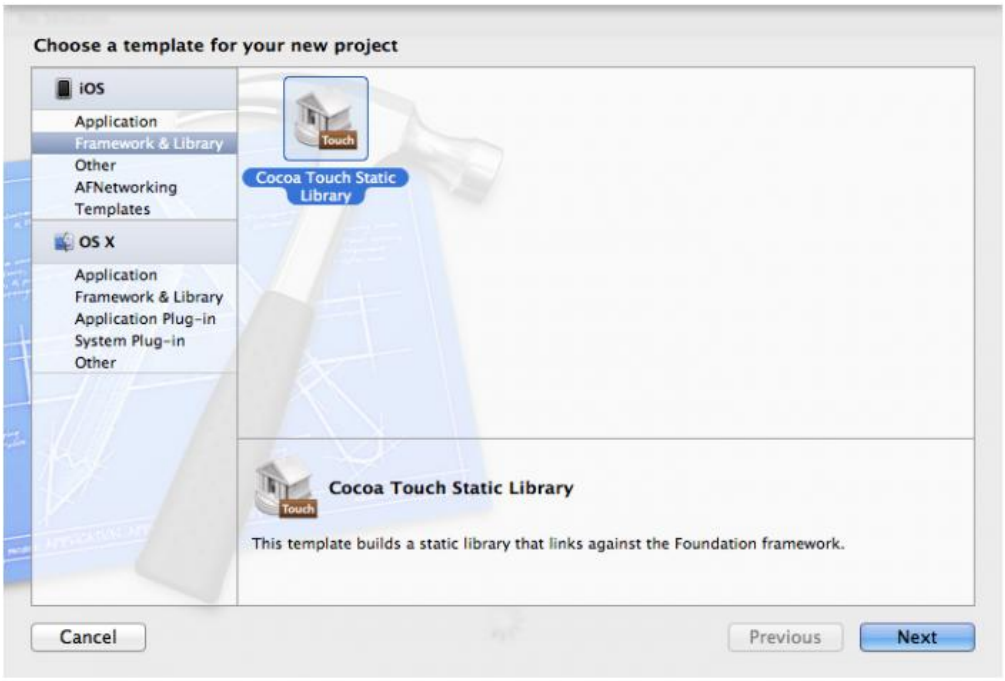
你想创建一个还在不断开发的库的快照版本。

本教程假设你已经完成学习 Core Image Tutorial，并对其中展示如何应用图片特效的代码得心应手。

将上述代码添加到一个静态库中，然后在一个 app 的修改版本中使用这个静态库。我们会得到一个带有上面列表中全部好处的完全相同的应用。

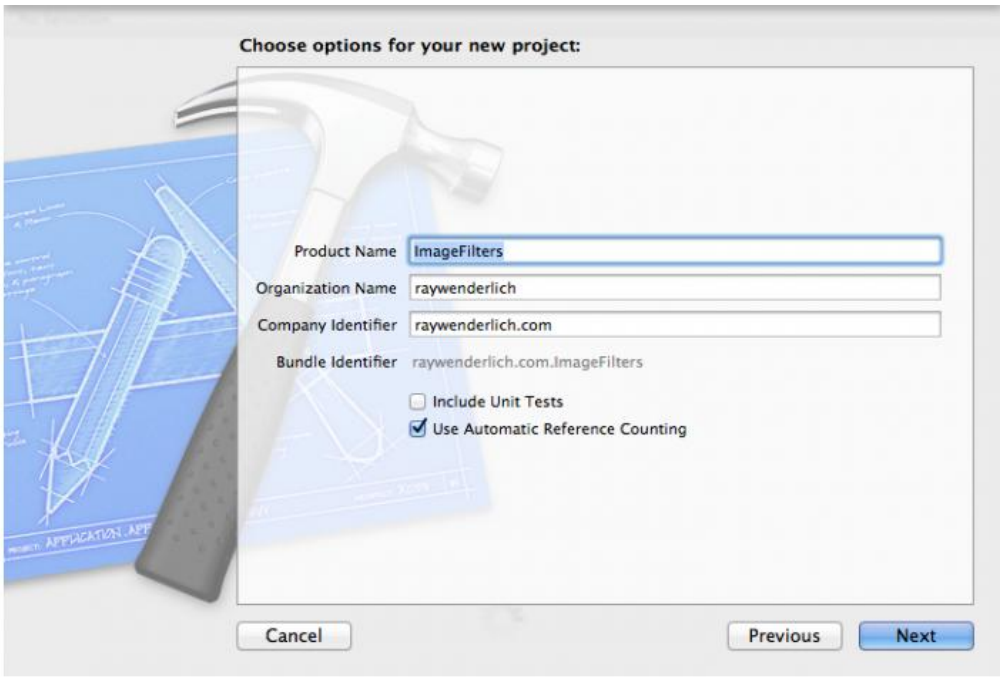
开始

运行 Xcode，选择 File\New\Project，在 Choose a template 对话框中选择 iOS\Framework & Library\Cocoa Touch Static Library,如下图:



New Lib

点击 Next。在工程选项对话框中，输入 ImageFilters 作为产品名。再输入一个唯一的公司标识，确保 Use Automatic Reference Counting 被选中且 Include Unit Tests 未选中。如下图:



Lib Name

点击 Next。最后，选择你想保存工程的位置并点击 Create。

Xcode 已经准备好静态库工程，甚至已经为你添加了一个 ImageFilters 类。这就是你的滤镜代码将要存放的地方。

注意：你可以添加任意数量的类到静态库中或者从中删除原有的类。本教程中的代码都会写在开始就被创建好的 ImageFilters 类中。

你的 Xcode 工程还是一片空白，现在我们添加一些代码进去！

图片滤镜

该库使用 UIKit，为 iOS 设计，所以你要做的第一件事就是在头文件中导入 UIKit。打开 ImageFilters.h，在文件顶部添加以下代码：

```
#import <UIKit/UIKit.h>
```

接下来将以下声明部分的代码粘贴到@interface ImageFilters : NSObject 下面

```
@property (nonatomic,readonly) UIImage *originalImage;

- (id)initWithImage:(UIImage *)image;
- (UIImage *)grayScaleImage;
- (UIImage *)oldImageWithIntensity:(CGFloat)level;
```

这些头文件中的声明定义了类的公开接口。其他开发者(包括你自己)使用该库时，只需通过阅读该头文件就可以知道类名和暴露的方法。

现在增加实现。打开 ImageFilters.m 文件，粘贴以下代码到#import "ImageFilters.h"下面：

```
@interface ImageFilters()

@property (nonatomic,strong) CIContext *context;
@property (nonatomic,strong) CIImage *beginImage;

@end
```

上面的代码声明了一些内部使用的属性。它们不是公开的，所以使用该库的引用没有使用它们的入口。

最后，你需要实现方法。粘贴以下代码到@implementation ImageFilters:下面：

```
- (id)initWithImage:(UIImage *)image
{
        self = [super init];
```

```
4.          if (self) {
5.                  _originalImage = image;
6.                  _context = [CIContext contex
   tWithOptions:nil];
7.                  _beginImage = [[CIImage alloc] in
   itWithImage:_originalImage];
8.          }
9.          return self;
10.  }
11.
12.  - (UIImage*)imageWithCIImage:(CIImage *)ciImage
13.  {
14.          CGImageRef cgiImage = [self.context createCGImage:
   ciImage fromRect:ciImage.extent];
15.          UIImage *image = [UIImage imageWithCGImage:cgiIma
   ge];
16.          CGImageRelease(cgiImage);
17.
18.          return image;
19.  }
20.
21.  - (UIImage *)grayScaleImage
22.  {
23.          if( !self.originalImage)
24.                  return nil;
25.
26.          CIImage *grayScaleFilter = [CIFilter filterWithNa
   me:@"CIColorControls" keysAndValues:kCIInputImageKey, self.be
   ginImage, @"inputBrightness", [NSNumber numberWithFloat:0.
   0], @"inputContrast", [NSNumber numberWithFloat:1.1], @"inp
   utSaturation", [NSNumber numberWithFloat:0.0], nil].outputIm
   age;
27.
28.          CIImage *output = [CIFilter filterWithName:@"CIEx
   posureAdjust" keysAndValues:kCIInputImageKey, grayScaleFilte
   r, @"inputEV", [NSNumber numberWithFloat:0.7], nil].outputI
   mage;
29.
30.          UIImage *filteredImage = [self imageWithCIImage:o
   utput];
31.          return filteredImage;
32.  }
33.
34.  - (UIImage *)oldImageWithIntensity:(CGFloat)intensity
```

```
35.  {
36.          if(  !self.originalImage  )
37.                  return  nil;
38.
39.          CIFilter  *sepia  =  [CIFilter  filterWithName:@"CISe
     piaTone"];
40.          [sepia  setValue:self.beginImage  forKey:kCIInputImag
     eKey];
41.          [sepia  setValue:@(intensity)  forKey:@"inputIntensit
     y"];
42.
43.          CIFilter  *random  =  [CIFilter  filterWithName:@"CIR
     andomGenerator"];
44.
45.          CIFilter  *lighten  =  [CIFilter  filterWithName:@"CI
     ColorControls"];
46.          [lighten  setValue:random.outputImage  forKey:kCIInpu
     tImageKey];
47.          [lighten  setValue:@(1  -  intensity)  forKey:@"input
     Brightness"];
48.          [lighten  setValue:@0.0  forKey:@"inputSaturation
     "];
49.
50.          CIImage  *croppedImage  =  [lighten.outputImage  imag
     eByCroppingToRect:[self.beginImage  extent]];
51.
52.          CIFilter  *composite  =  [CIFilter  filterWithName:@"
     CIHardLightBlendMode"];
53.          [composite  setValue:sepia.outputImage  forKey:kCIInp
     utImageKey];
54.          [composite  setValue:croppedImage  forKey:kCIInputBac
     kgroundImageKey];
55.
56.          CIFilter  *vignette  =  [CIFilter  filterWithName:@"C
     IVignette"];
57.          [vignette  setValue:composite.outputImage  forKey:kCI
     InputImageKey];
58.          [vignette  setValue:@(intensity  *  2)  forKey:@"inpu
     tIntensity"];
59.          [vignette  setValue:@(intensity  *  30)  forKey:@"inp
     utRadius"];
60.
61.          UIImage  *filteredImage  =  [self  imageWithCIImage:v
     ignette.outputImage];
```

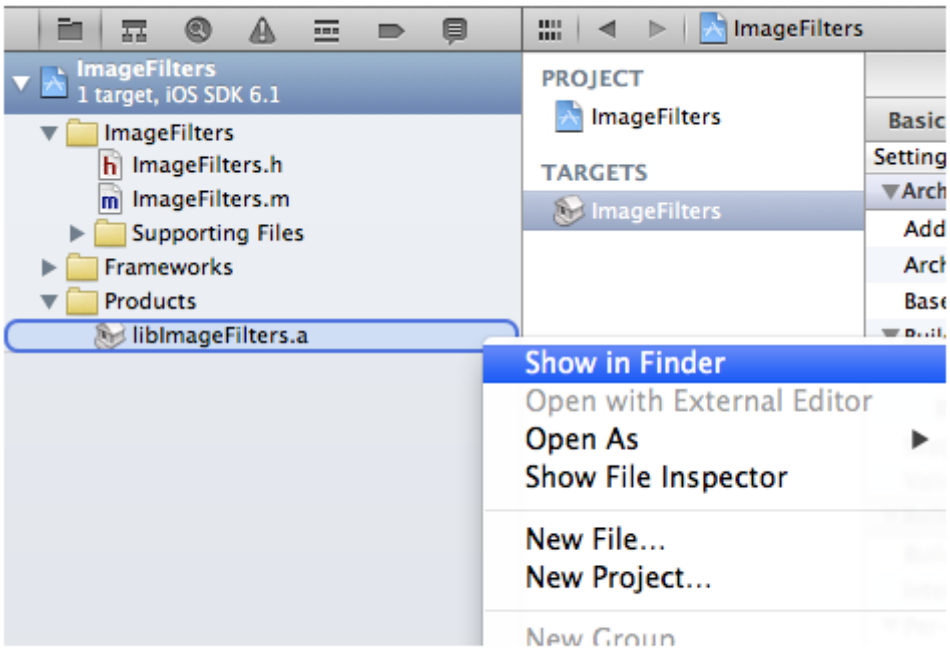
```
62.
63.         return  filteredImage;
64.  }
```

这段代码实现了初始化和图片滤镜功能。详细解释上述代码的功能已经超出了本教程的范围，你可以从 Core Image Tutorial 中了解到更多的关于 Core Image 和滤镜的知识。

到这里，你已经有了一个静态库，它有一个暴露了以下 3 个方法的公开类 Image Filters:
initWithImage : 初始化滤镜类
grayScaleImage : 创建灰阶图片
oldImageWithIntensity : 创建怀旧效果的图片

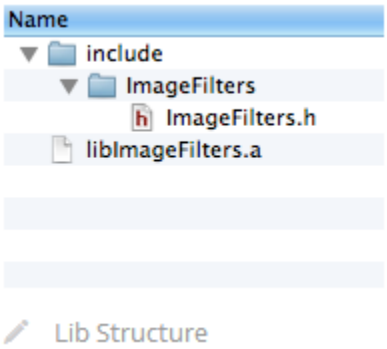
现在构建并运行你的库。你会注意到 Xcode 的”Run”按钮只是执行了一次构建，而并不能真正的运行库去查看效果，因为并没有真正的 app。

静态库的后缀名是.a 而并不是一个.app 或者.ipa 文件。可以在工程导航栏中的 Products 文件夹下找到生成的静态库。右键点击 libImageFilters.a 并在弹出菜单中选择 Show in Finder。



Show in Finder

Xcode 会在 Finder 中打开文件夹，你可以看到以下类似的结构:



离完成一个库产品还剩两件事:
1.Header files ： 在 include 文件夹中可以找到库的所有公开头文件。在该示例中，只有一个公开类所以文件夹中只有一个 ImageFilters.h 文件。稍后你会在你的 app 工程中用到这个头文件以便于 Xcode 在编译期识别暴露的类。
2.Binary Libraty ： Xcode 生成的静态库是 ImageFilters.a。想在应用中使用该库，你需要用该文件链接。

这两个部分和你想在应用里包含一些新的框架时所需要做的事很相似，简单的导入框架头文件并建立链接。

库已经准备就绪，需要附加说明的是，默认情况下，库文件只会为当前的架构构建。如果你在模拟器下构建，那么库会包含对应 i386 架构的结果代码；如果在真机设备下构建，你将会得到对应 ARM 架构的代码。你可能需要构建两个版本的库，并且当从模拟器切换到设备的时候选择其中一个使用。

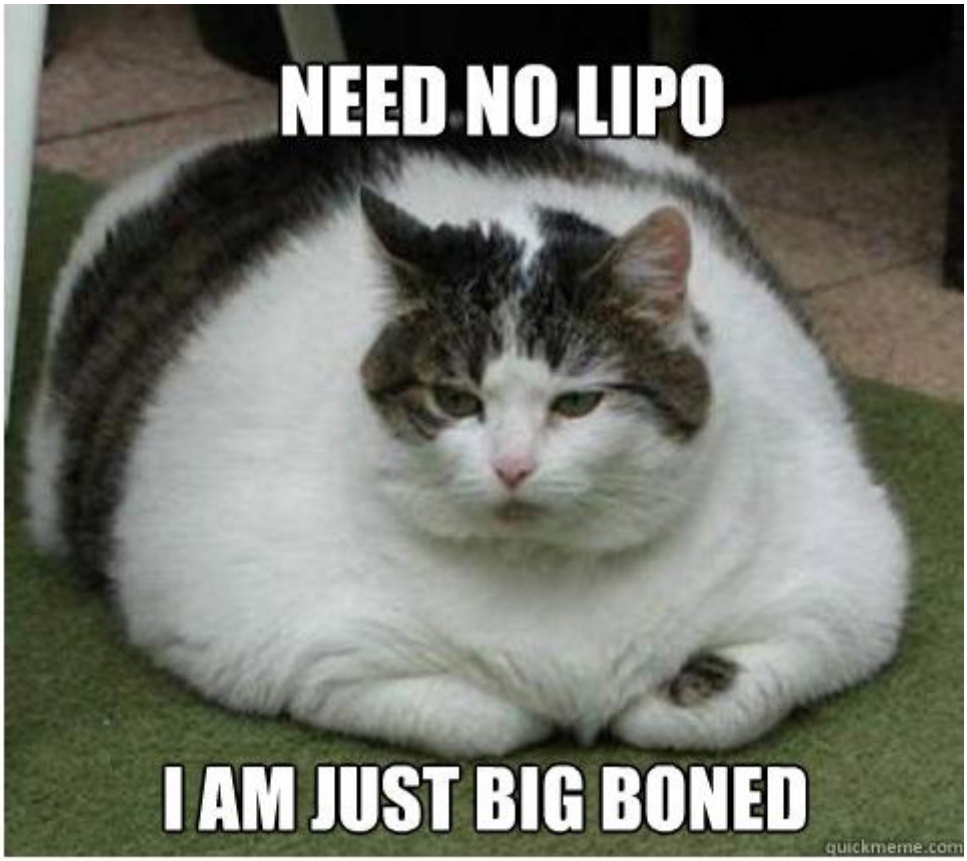
怎么办?

幸运的是，有一个更好的办法可以不建立多个配置或在工程中构建产品就可以支持多个平台。你可以创建一个对应 2 个 架构的包含结果代码的 universal binary。

通用二进制

通用二进制是一种特殊的二进制文件，它包含对应多个架构的结果代码。你可能在从 PowerPC(PPC)到 Inter(i386)的 Mac 电脑产品线的过渡中对其有所熟悉。在这个过程中，Mac 应用程序通常迁移为包含 2 个 可执行包的一个二进制文件，这样应用程序即能在 Inter 也能在 PowerPC 的 Mac 电脑上运行。

同时支持 ARM 和 i386 的概念并没有太大不同。在这里静态库要包含支持 iOS 设备(ARM)和模拟器(i386)的结果代码。Xcode 可以识别通用库，每次你构建应用的时候，它会根据目标选择适当的架构。
为了创建通用二进制库，需要使用一个名为 lipo 的系统工具。



Lipo Cat

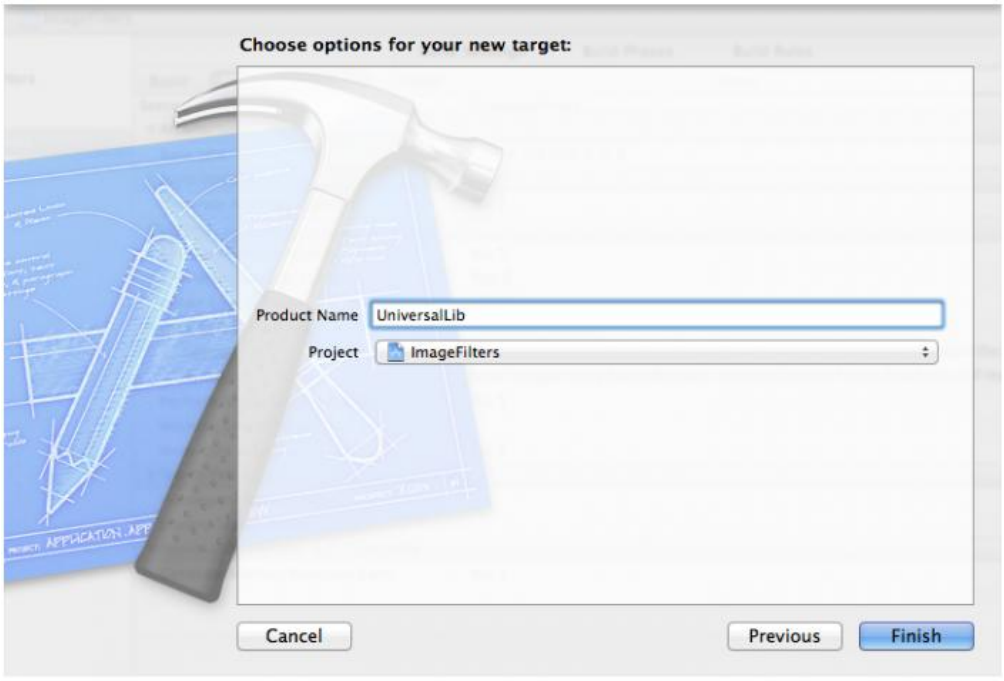
别担心，不是那种 lipo！:] (lipo 有脂肪的意思 — 译者注)

lipo 是一个命令行工具，它允许在通用文件上执行操作(类似于创建通用二进制，列出通用文件内容等等)。本教程中使用 lipo 的目的是联合不同架构的二进制文件到单个输出文件中。你可以直接在命令行中使用 lipo 命令，但在本教程中你可以让 Xcode 执行一段创建通用库的命令行脚本来为你做这件事。

Xcode 中一个集合目标可以一次构建多个目标，包括命令行脚本。在 Xcode 菜单中选择 File/New/Target，选择 iOS/Other 并点击 Aggregate，如图:

Aggregate Target

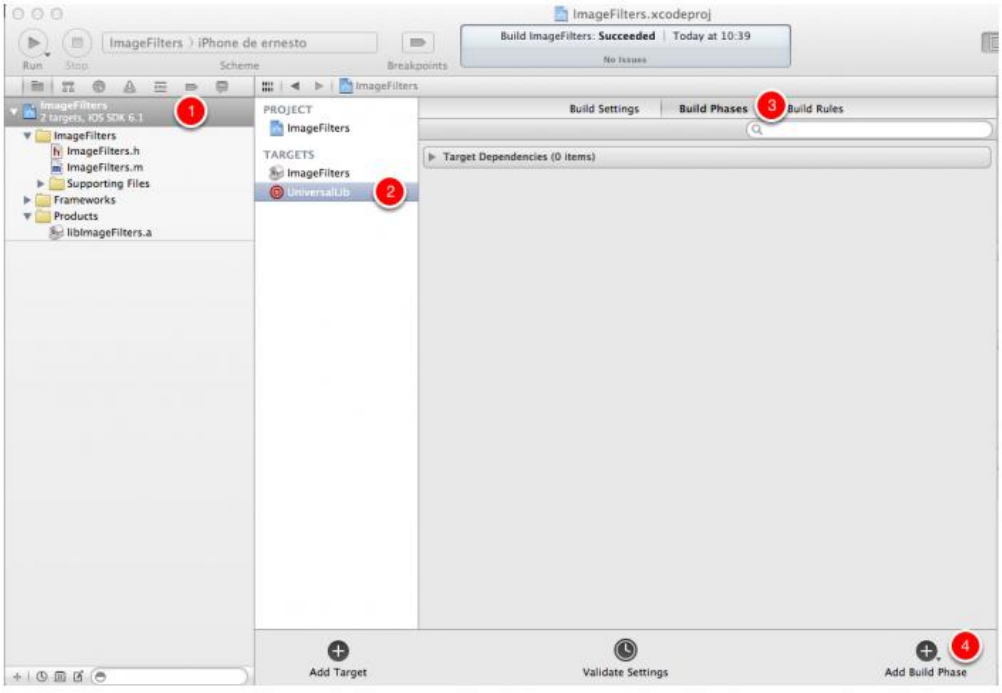
将目标命名为 UniversalLib，确保选中 ImageFilters 工程，如图:



Aggregate Universal

在工程导航视图中选中 ImageFilters，然后选择 UniversalLib 目标。切换到 Build Phases 标签；在这里设置构建目标时将要执行的动作。

点击 Add Build Phase 按钮，在弹出的菜单中选择 Add Run Script，如下图:



Aggregate Phase

现在你需要设置脚本项。展开 Run Script 模块，在 Shell 行下粘贴如下代码:

```
1.  # define output folder environment variable
2.  UNIVERSAL_OUTPUTFOLDER=${BUILD_DIR}/${CONFIGURATION}-universa
    l
3.
4.  # Step 1. Build Device and Simulator versions
5.  xcodebuild -target ImageFilters ONLY_ACTIVE_ARCH=NO -conf
    iguration ${CONFIGURATION} -sdk iphoneos BUILD_DIR="${BUI
    LD_DIR}" BUILD_ROOT="${BUILD_ROOT}"
6.  xcodebuild -target ImageFilters -configuration ${CONFIGUR
    ATION} -sdk iphonesimulator -arch i386 BUILD_DIR="${BUILD_
    DIR}" BUILD_ROOT="${BUILD_ROOT}"
7.
8.  # make sure the output directory exists
9.  mkdir -p "${UNIVERSAL_OUTPUTFOLDER}"
10.
11. # Step 2. Create universal binary file using lipo
12. lipo -create -output "${UNIVERSAL_OUTPUTFOLDER}/lib${PROJE
    CT_NAME}.a" "${BUILD_DIR}/${CONFIGURATION}-iphoneos/lib${PROJE
    CT_NAME}.a" "${BUILD_DIR}/${CONFIGURATION}-iphonesimulator/lib
    ${PROJECT_NAME}.a"
13.
14. # Last touch. copy the header files. Just for conven
    ience
15. cp -R "${BUILD_DIR}/${CONFIGURATION}-iphoneos/include" "
    ${UNIVERSAL_OUTPUTFOLDER}/"
```

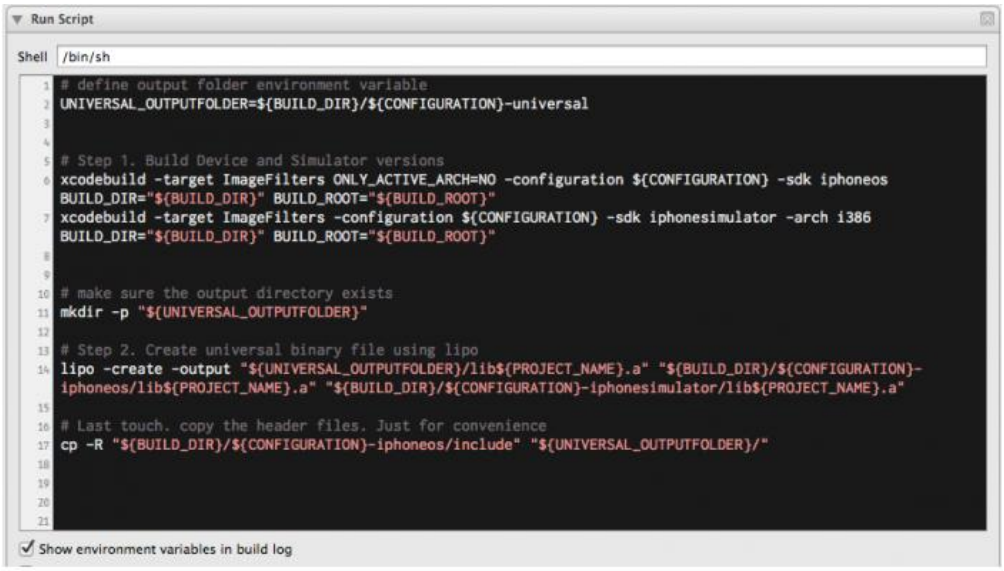
代码并不十分复杂，它是这样工作的:
UNIVERSAL_OUTPUTFOLDER 包括了通用二进制包将要被存放的文件夹:“Debug-universal”

Step 1. 第 2 行执行了 xcodebuild 并命令它构建 ARM 架构的二进制文件。(你可以看到这行中的-sdk iphoneos 参数)
下一行再次执行了 xcodebuild 命令并在另一个文件夹中构建了一个针对 Inter 架构的 iPhone 模拟器的二进制文件,在这里关键参数是-sdk iphonesimulator -arch i386。(如果感兴趣，你可以在 man page 了解更多关于 xcodebuild 的资料)

Step 2. 现在已经有了 2 个.a 文件分别对应两个架构。执行 lipo -create，用它们创建出一个通用二进制。

最后一行的作用是复制头文件到通用构建文件夹的外层。(用 cp 命令)

你的 Run Script 窗口应该看起来如下:



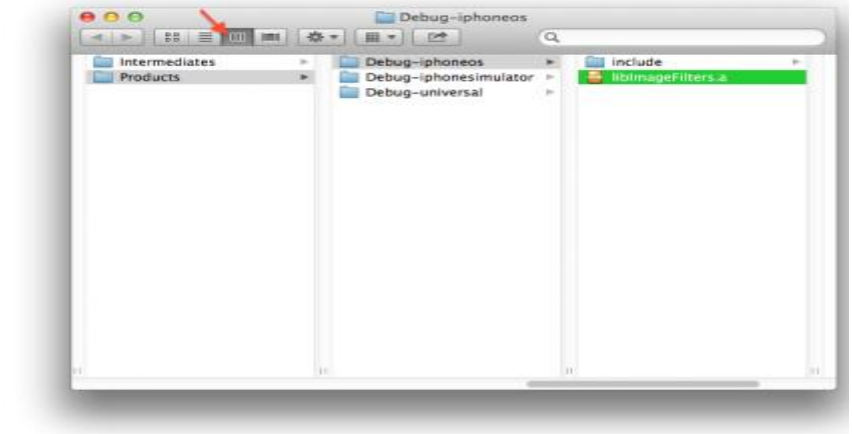
Aggregate Script

现在你已经准备好构建一个静态库的通用版本。在方案下啦菜单中选择集合目标 UniversalLib，如下(不像截图上的”iOS Device”，你看到的可能是自己的设备名字):

Aggregate Scheme

点击 Play 按钮来为集合方案构建目标。

在 libImageFilters.a 上再次选择 Show in Finder 查看结果。将 Finder 切换到列视图查看文件夹层次，可以看到一个包含库的通用版本的叫做 Debug-Universal 的新文件夹(或 Release-Universal 如果你构建了发布版本)，如下图:



St Finder

除了这个链接到模拟器和真实设备的二进制文件，你还可以找到普通的头文件和静态库文件。

这是你创建自己的通用静态库所需要学习的所有知识。

概括起来，一个静态库工程和一个应用工程非常相似。可以拥有一个或多个类，最后的产品是头文件和一个.a 文件。这个.a 文件就是可以链接到多个应用程序中的静态库。

在应用中使用静态库

在应用中使用 ImageFilters 类和直接使用源代码并没有太大区别:导入头文件然后开始使用类。问题是 Xcode 并不知道头文件和库文件的位置。

有两种办法可以将静态库引入到工程中:
方法 1: 直接引用头文件和库二进制文件(.a)
方法 2: 将库工程作为子项目

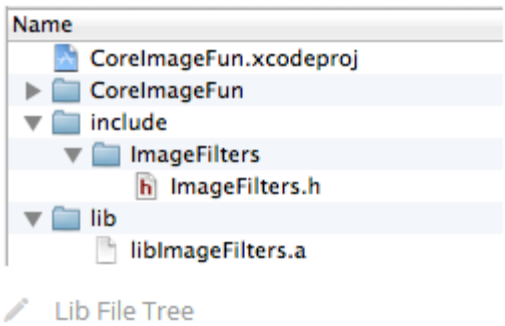
选择哪一种方法完全取决于你的喜好或者是否有静态库的源代码和工程配置文件任由你支配。

本教程将分别介绍两种方法。你可以自由尝试第一个或第二个，但推荐按照文中介绍的顺序分别尝试两个。在两个部分的开头，需要一个 zip 文件，该文件是在 Core Image Tutorial 中创建的应用的修改版本，修改后的版本使用了库中新的 ImageFilters 类。

本教程的主要目的是教你如何使用静态库，所以修改后的工程包括了所有应用需要的源代码。这样你就可以将注意力集中在使用库所需要的工程设置上。

方法 1: 头文件和库二进制文件

在本节中，你需要下载 starter project for this section。复制压缩文件到硬盘上的任意文件夹并解压。可以看到如下的文件夹结构:

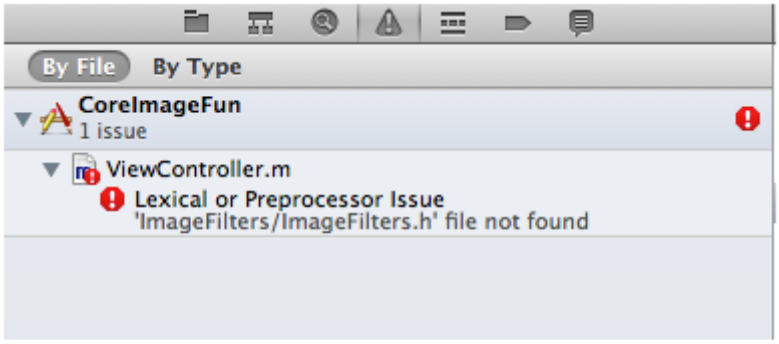


Lib File Tree

为了方便起见，.a 通用库文件和头文件已经复制了一份在其中，但工程并未设置使用它们。你将从这里开始。

注意：标准的 Unix 引入惯例是一个 include 文件夹，用来存放头文件，一个 lib 文件夹用来存放库文件(.a)。这种文件夹结构这是一种惯例，并不强制。你并不需要一定遵从这种结构或者复制文件到工程文件夹中。在你自己的应用中，你可以任意选择头文件和库文件的位置，只要随后在 Xcode 工程中设置了适当的路径。

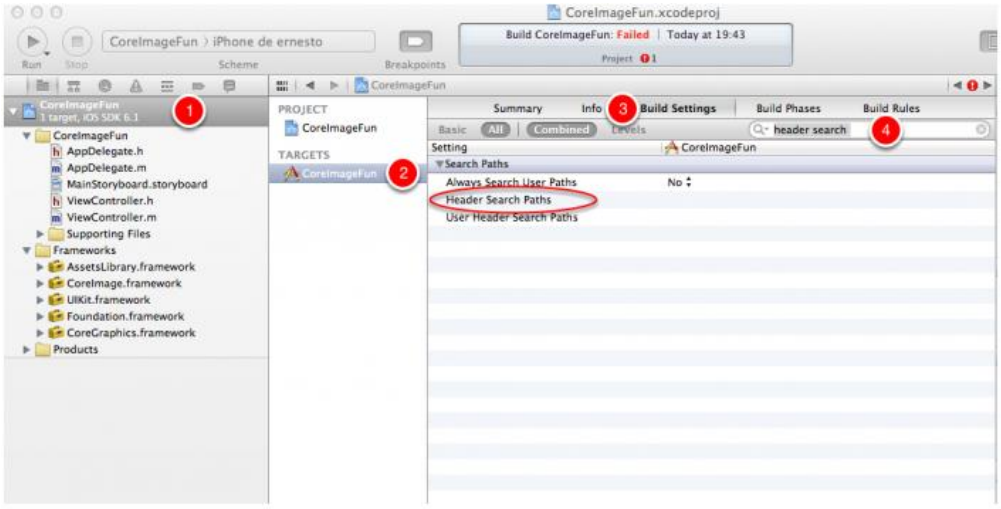
打开工程，构建并运行你的应用，将会看到以下错误：



Lib Error Include

正如所期望的那样，应用并不知道去哪里寻找头文件。为了解决这个问题，你需要在工程中添加一个 Header Search Path，指明头文件存放的位置。设置头文件搜索路径始终是使用静态库的第一步。

按照下图示范，在导航栏中点击工程根节点(1)，选择 CoreImageFun 目标(2)。选择 Build Settings(3)，在列表中找出 Header Search Paths 设置项。如果必要，可以在搜索框中输入”header search”来过滤庞大的设置列表(4)。

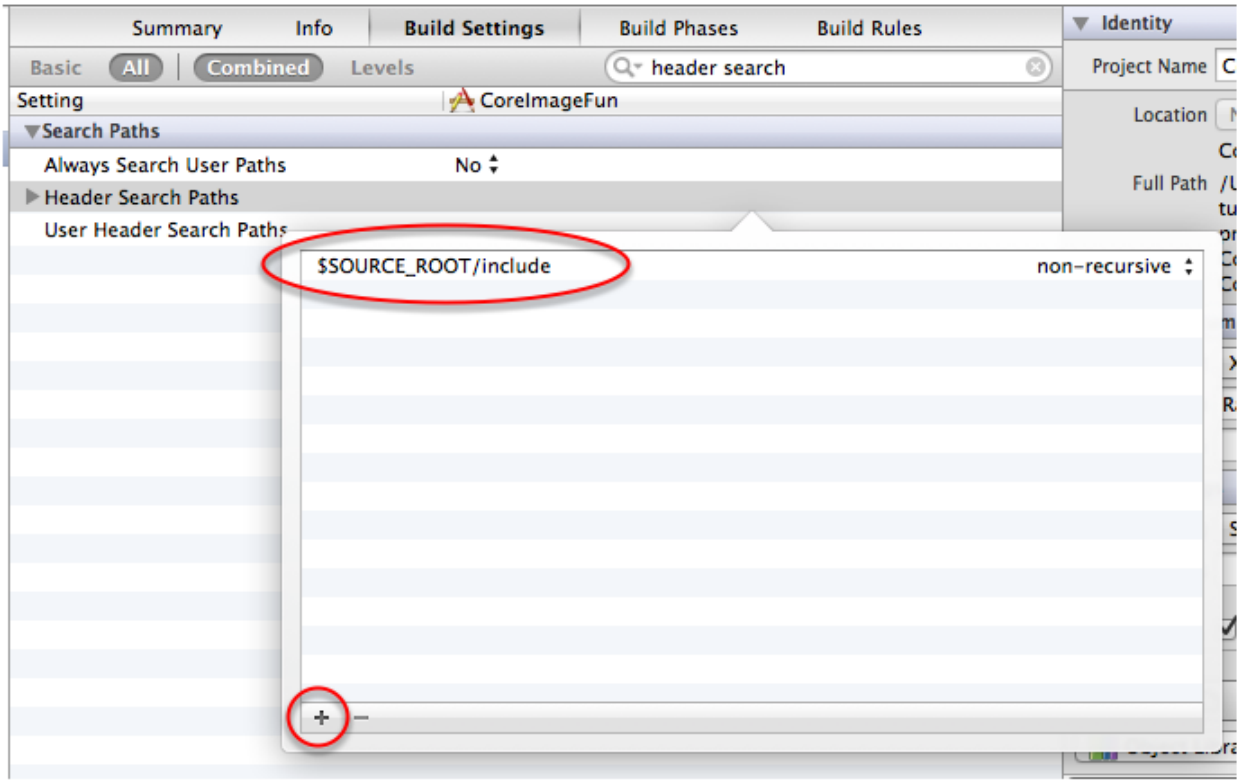


Lib Header Search

双击 Header Search Paths 项，弹出一个浮动窗口，点击+按钮，输入：

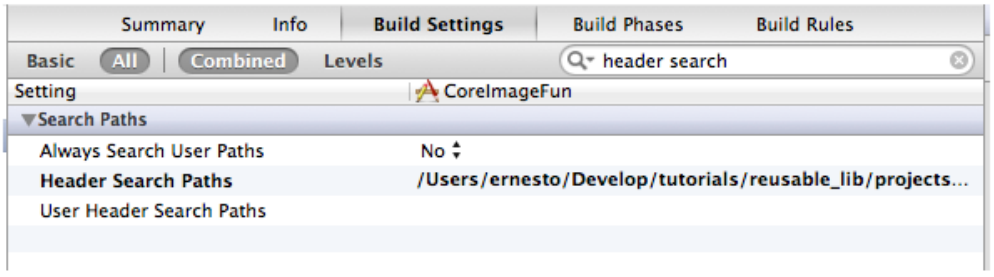
```
1.    $SOURCE_ROOT/include
```

弹出窗口应该如下所示：



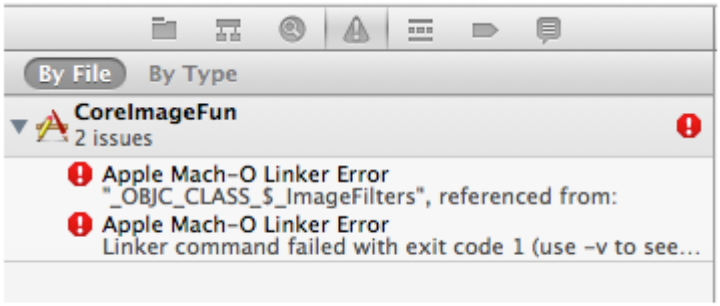
Lib Header Search

$SOURCE_ROOT 是一个 Xcode 环境变量，指向工程根文件夹。Xcode 会使用包含你工程的实际文件夹代替此变量，这意味着即使你把工程移动到其它文件夹或驱动器，它仍然可以指向最新的位置。
在弹出窗口范围外点击鼠标使其消失，你会看到 Xcode 已经自动将变量转换为工程的实际位置，如图所示：



Lib Header Search

构建并运行应用，看看结果是什么。呃……一些链接错误出现了：

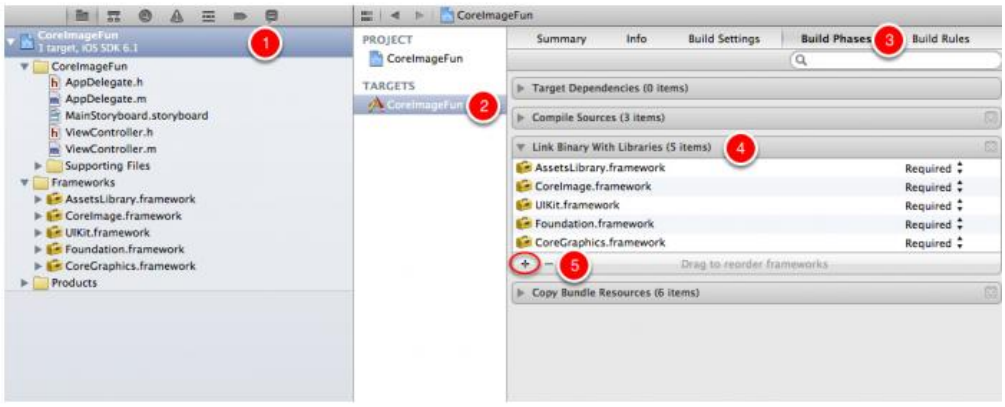


Lib Header Search

这看起来并不是很好，但是给了你另一个你所需要的信息。仔细看，会发现所有的编译错误全都消失了，全部被链接错误所代替。这表示 Xcode 找到了头文件并且用它去编译应用，但在链接阶段，Xcode 无法找到 ImageFilter 类的结果代码。为什么？

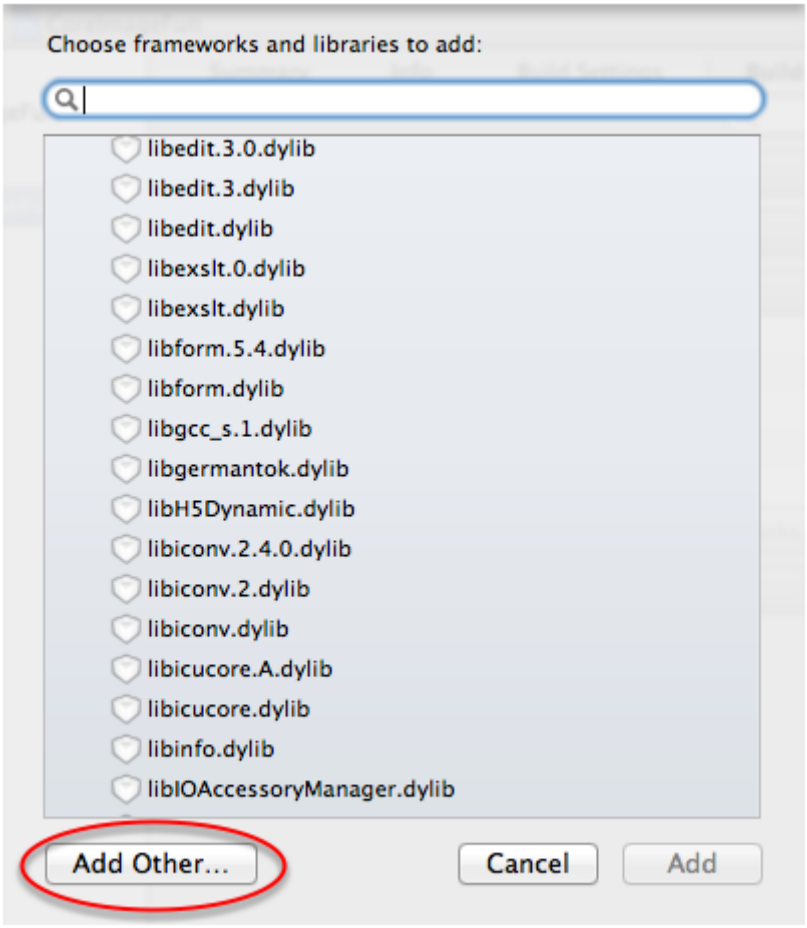
很简单 — 你还没有告诉 Xcode 去哪里寻找包含类实现的库文件。(看，没什么大不了)

如下面的屏幕截图所示，回到 CoreImageFun 目标(2)的构建设置(1)。选择 Build Phases 标签(3)，展开 Link Binary With Libraries 部分(4)。最后，点击+按钮(5)。



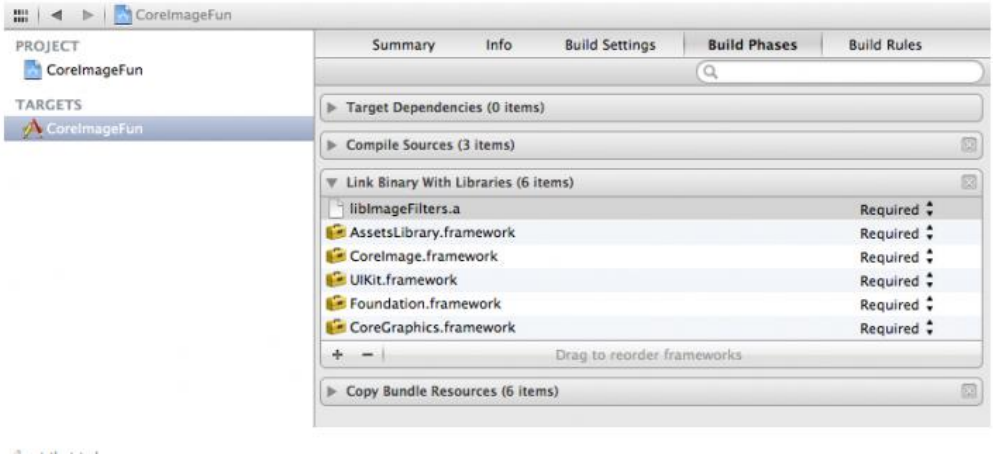
Lib Link

在出现的窗口中，点击 Add Other…按钮，在工程根文件夹下的 lib 子目录中找到 libImageFilters.a 库文件，如图:



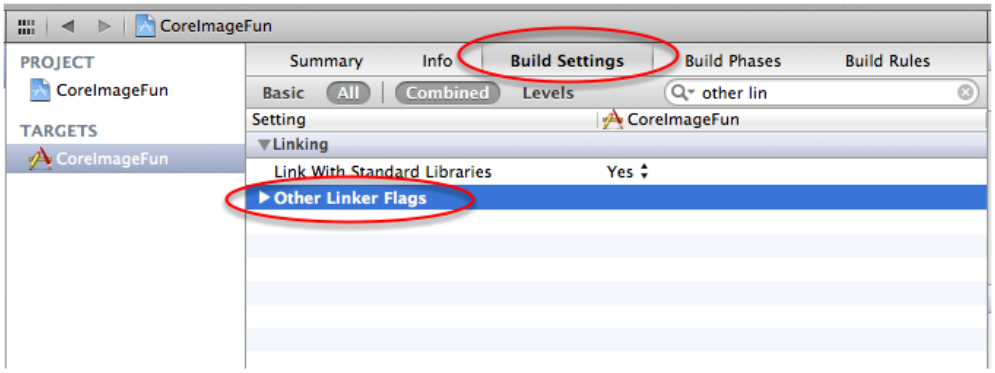
Lib Link

完成这些以后，你的 Build Phase 标签看起来如下：



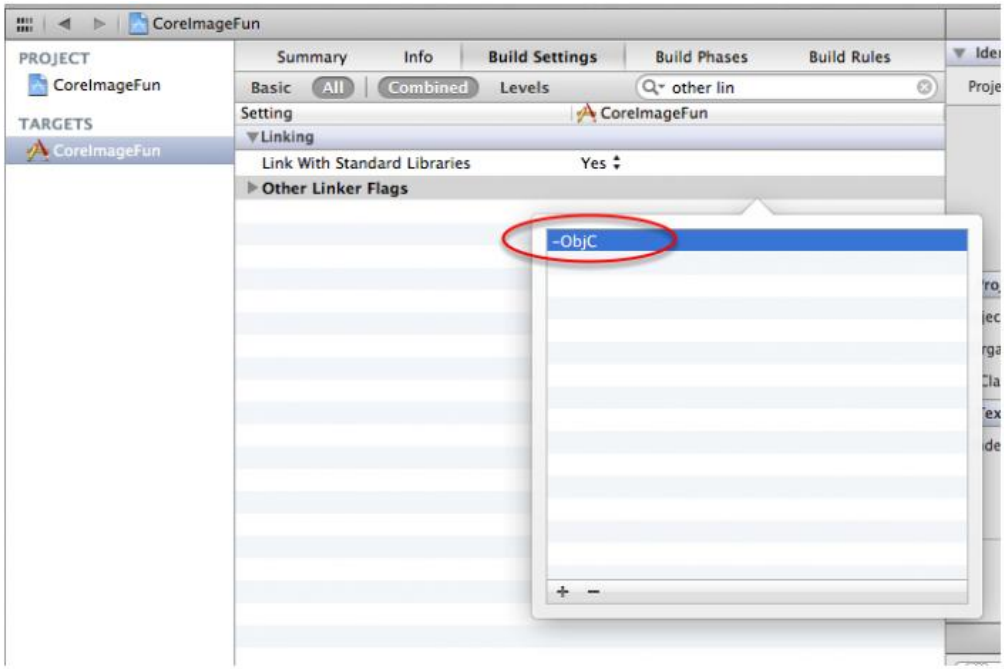
最后一步是增加-ObjC 链接标识。该链接尝试更高效的只包含需要的代码，而有时会排除静态库代码。使用该标识，库中的所有 Objective-C 类和类别都将被适当的加载。你可以从苹果的 [Technical Q&A QA1490](#) 了解详细信息。

点击 Build Settings 标签，找到 Other linker Flags 设置，如图：



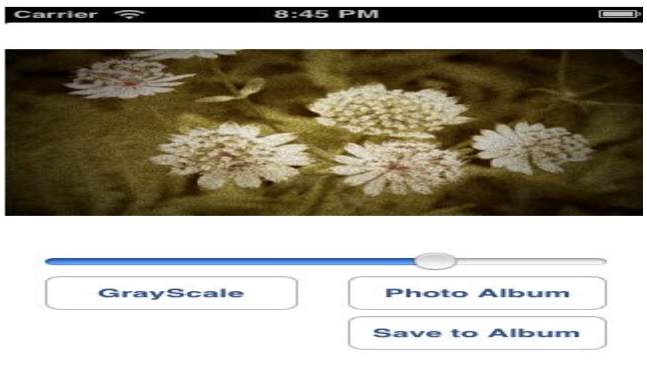
OBJC Flag

在弹出窗口中，点击+按钮并输入-ObjC，如图：



OBJC Flag

最后构建并运行应用，此时应该不会得到任何构建错误信息，应用顺利展示它的光彩之处：

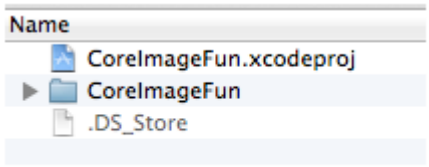


拖动滑块改变滤镜级别，或者点击 GrayScale 按钮。对图片应用滤镜的代码来自于静态库，而不是应用。

恭喜 — 你已经构建了你的第一个静态库并在一个真正的应用里使用它!你会发现这种包含头文件和库的方法在很多第三方库中使用，如 AdMob，TestFlight 或一些不提供源代码的商业库。

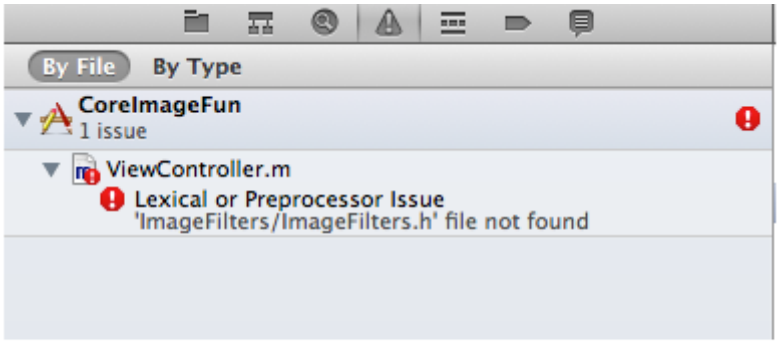
方法 2：子项目
在这部分，请在这里下载所需工程。

复制下载的文件到任意位置，解压。可以看到以下文件夹结构：



Subproject Structure

如果学习了方法一，你可能注意到了工程的差异。这个工程里没有任何的头文件和静态库文件 — 因为根本不需要。作为替代方案，你要将你在本教程开始创建的 ImageFilters 库工程添作为依赖加到本工程中。
在做这些之前，构建并运行应用。会看到以下错误：

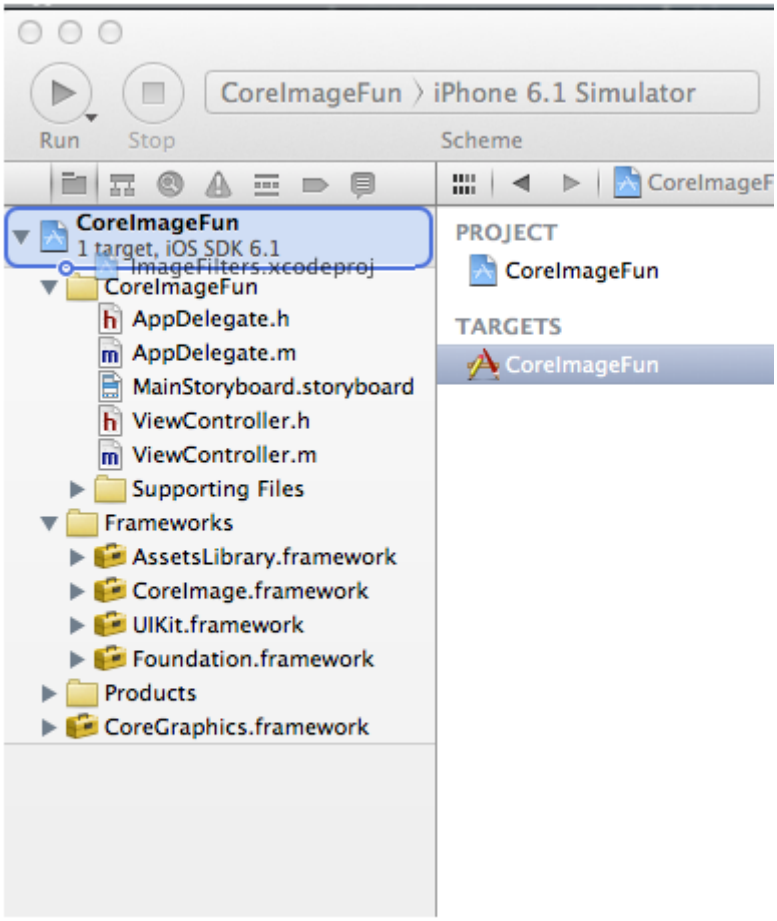


Lib Error Include

如果学习过上一个方法，你已经知道如何修复这个问题。在示例工程中，你在 ViewController 类中使用了 ImageFilters 类，但并未告诉 Xcode 去哪里寻找头文件。Xcode 会尝试寻找 ImageFilters.h 文件，但是失败了。

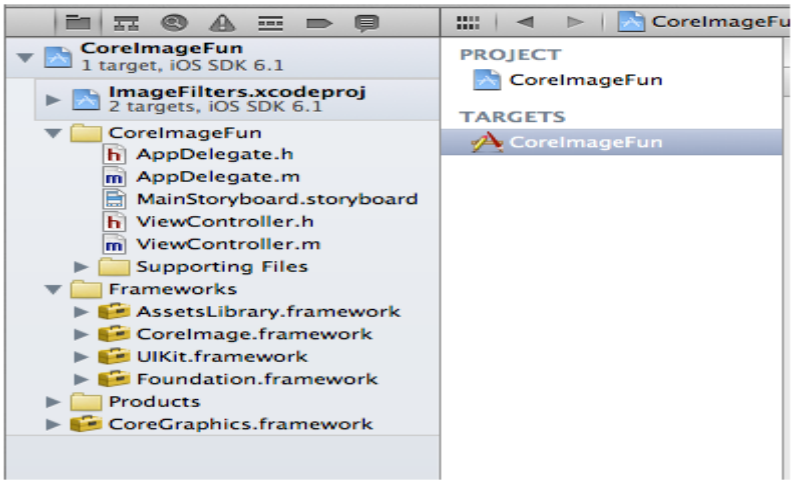
将 ImageFilters 库工程作为子项目所需的所有操作就是拖拽库工程文件到库文件树中。如果该工程已经在另一个 Xcode 窗口中被打开，那么 Xcode 无法正确将其添加为子工程。所以在继续本教程之前，确保 ImageFilters 库工程已经被关闭。

在 Finder 中找到名为 ImageFilters.xcodeproj 库工程文件。拖拽它到 CoreImageFun 工程左侧的导航栏中，如图：



Subproject Drag

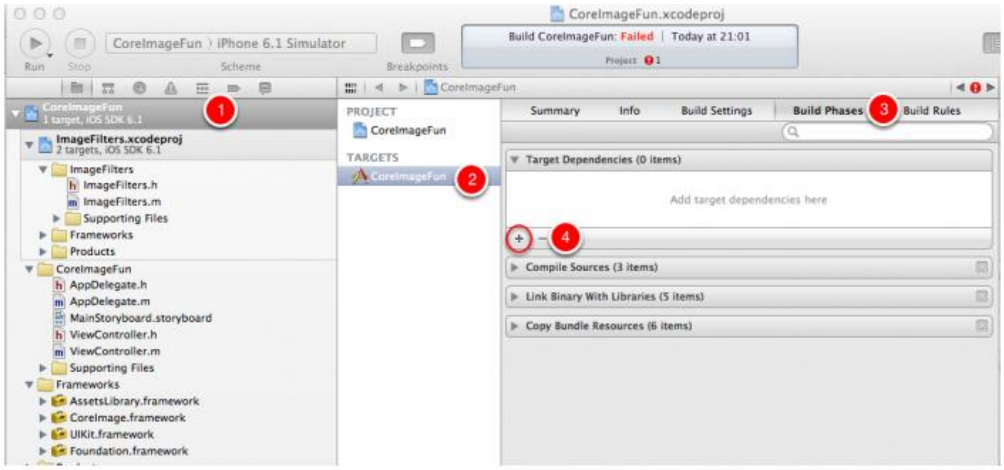
完成拖放后，你的工程浏览视图应该如下图所示：



Subproject Drag

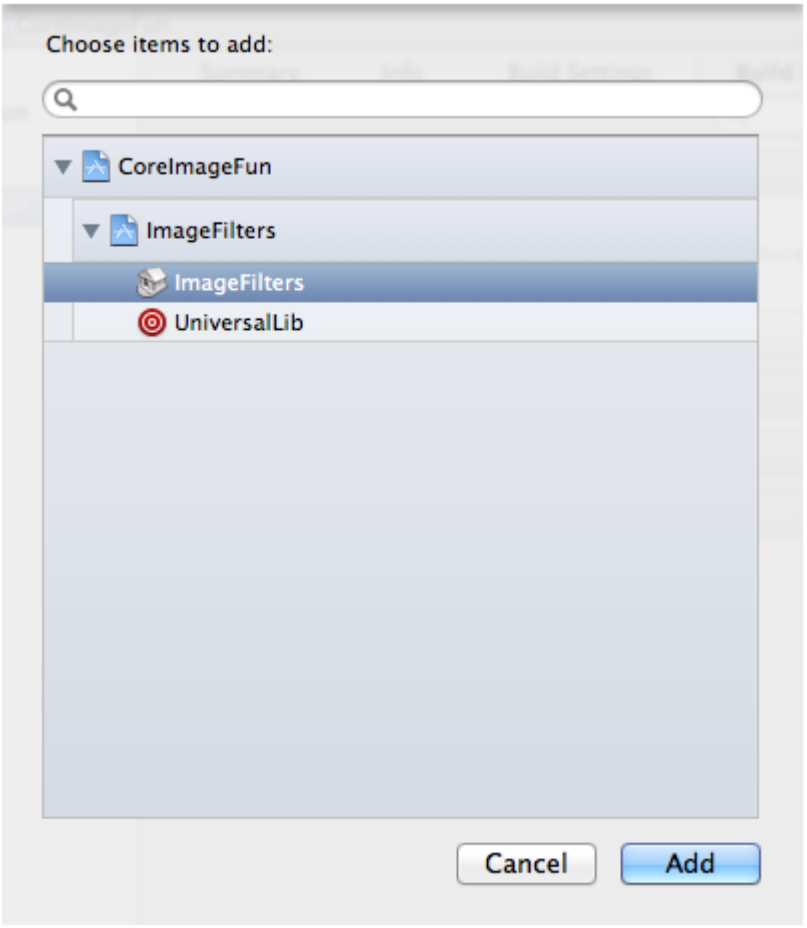
现在 Xcode 已经识别了子工程，你可以将库添加为工程依赖。这样 Xcode 就可以在构建主应用之前确保库为最新版本。

点击工程文件(1)，选择 CoreImageFun 目标(2)。点击 Build Phases 标签(3)并展开 Target Dependencies(4)，如图:



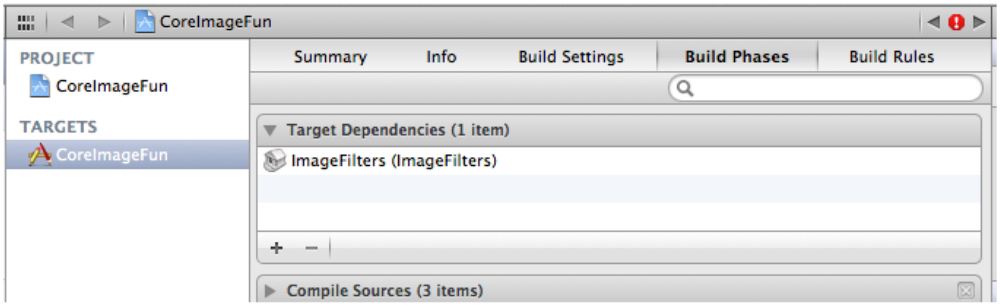
Dependencies

点击+按钮增加一个新依赖。如下图所示，确保你从弹出窗口中选择了 ImageFilters 目标(不是 universalLib):



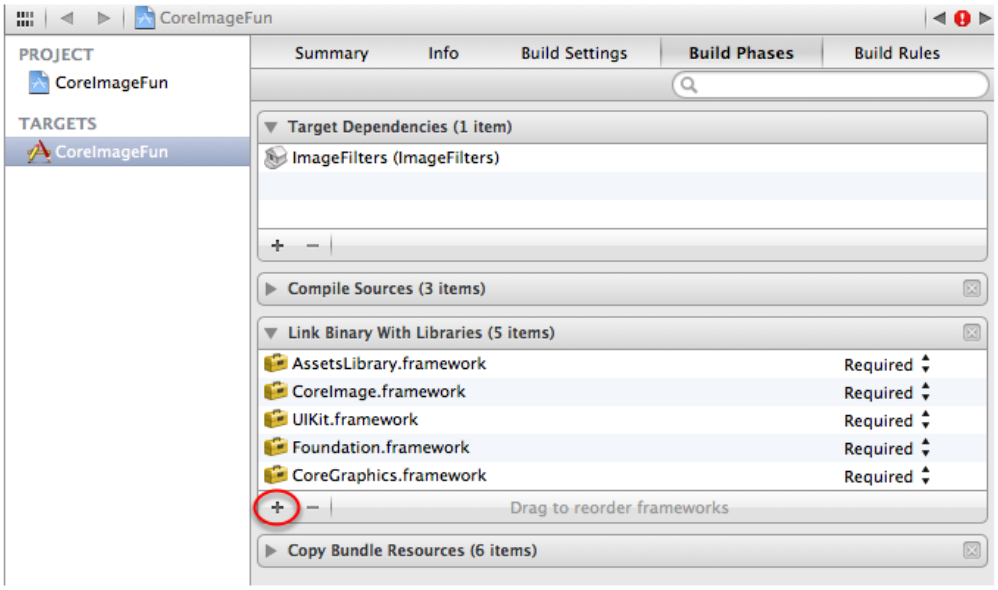
Dependency

添加完成之后，依赖窗口应该如图所示：



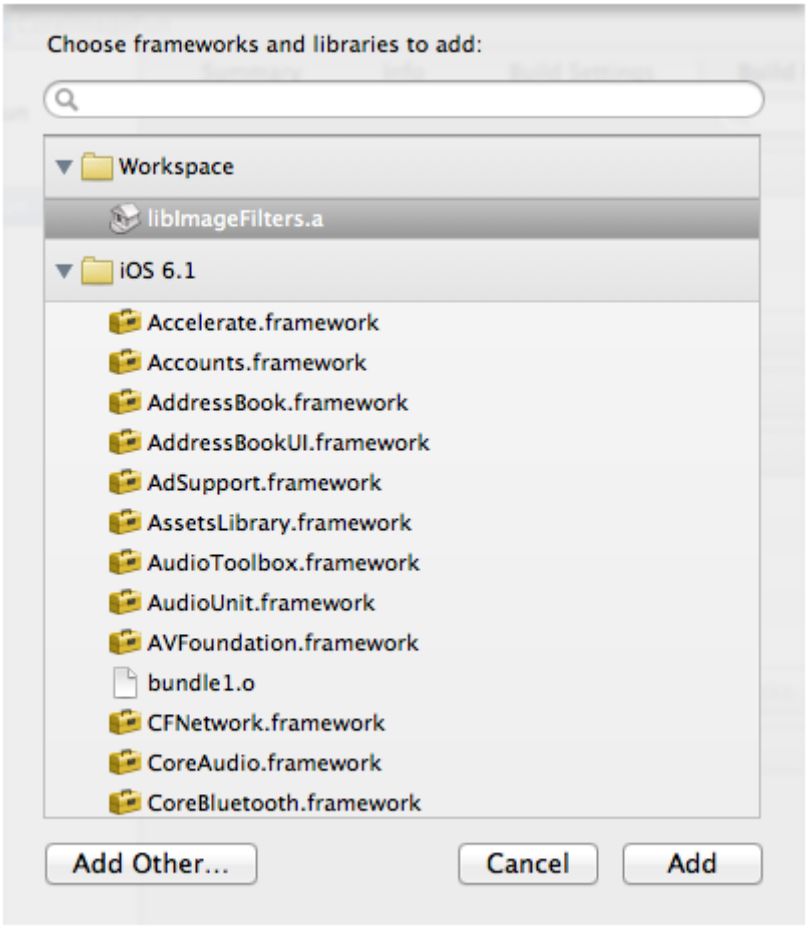
Dependency

最后，设置静态库工程链接到应用。展开 Link Binary with libraries，点击+按钮，如图：



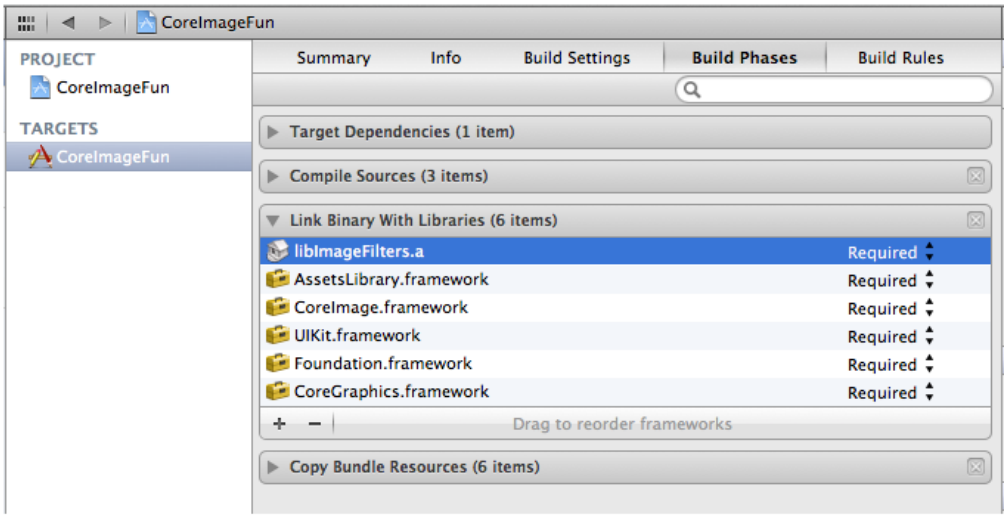
Subproject Link

选择 libImageFilters.a，点击 Add：



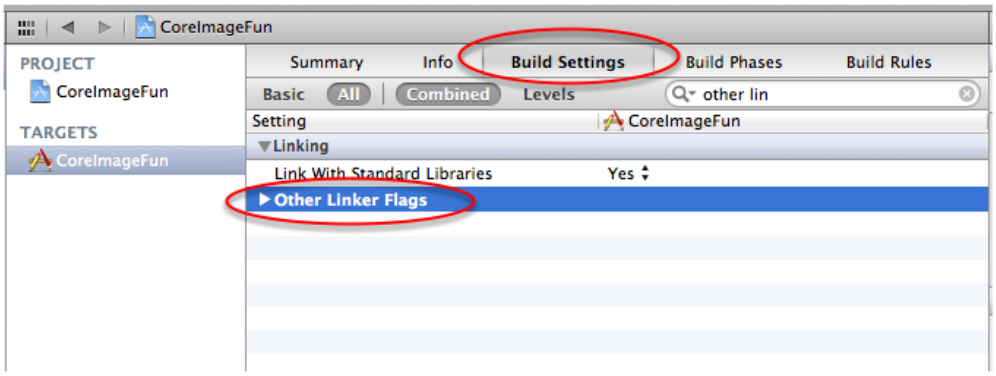
Subproject Link

添加库之后，Link Binary with Libraries 部分应该如图所示：



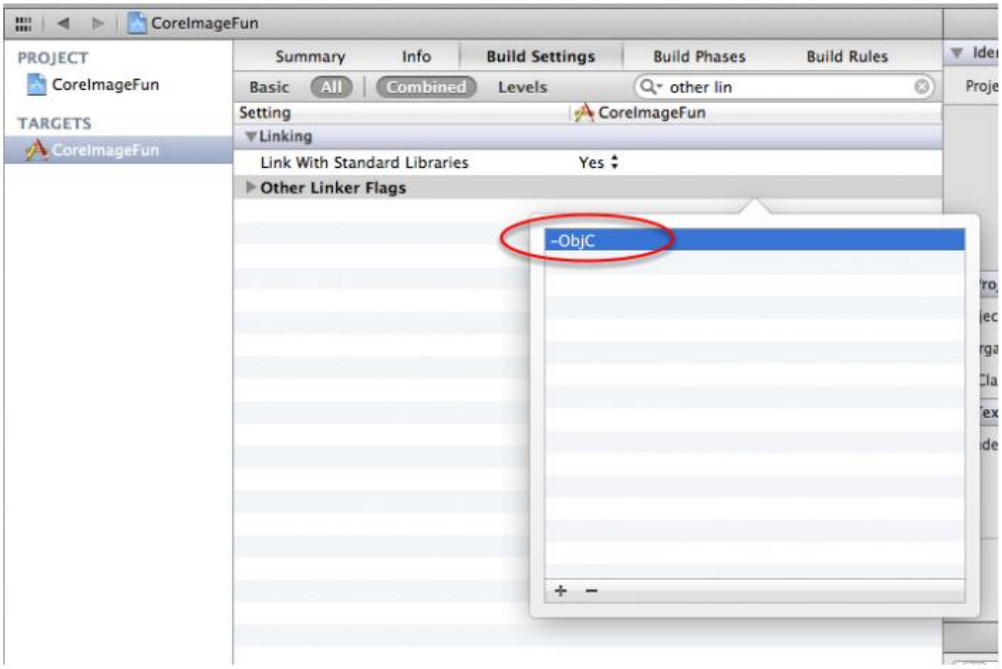
Subproject Link

像方法一那样，最后一步是增加-ObjC 链接标识。点击 Build Settings 标签，找到 Other linker Flags 设置，如图：



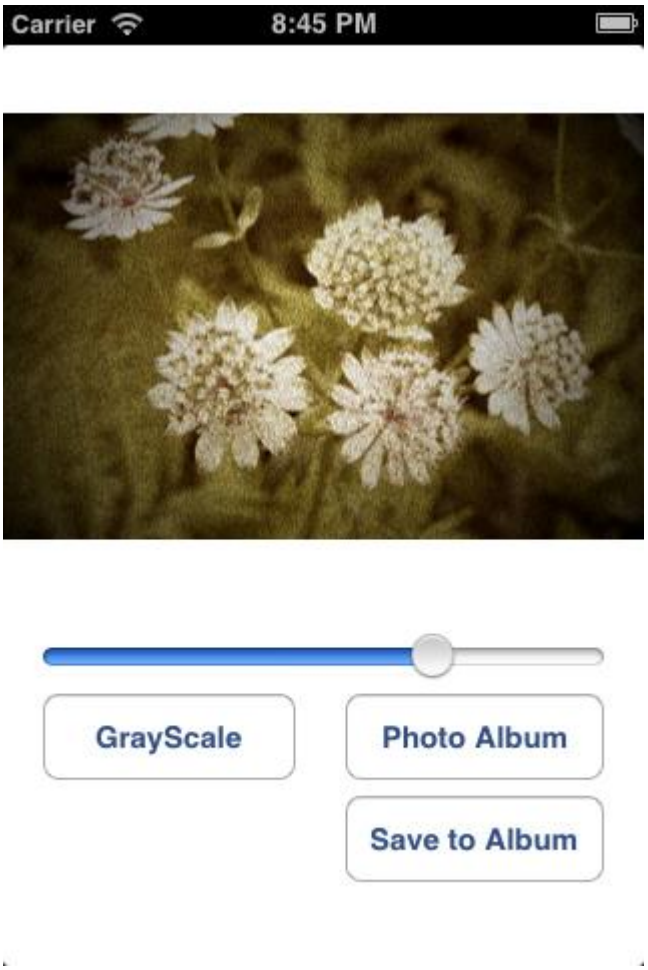
OBJC Flag

在弹出窗口中，点击+按钮并输入-ObjC，如图：



OBJC Flag

构建并运行应用，应该没有任何错误，应用会再一次被打开：



拖动滑块或者点击 GrayScale 按钮查看图片滤镜结果。滤镜逻辑的代码完全包含在库中。

如果按照第一种方法在应用中添加库(使用头文件和库文件)，你可能注意到和第二种方法的区别。在方法二中，你没有在工程设置中添加任何头文件搜索路径。另一个区别是你没有使用通用库。

为什么会有这样的区别？当添加一个库作为一个子工程，Xcode 会为你考虑几乎所有的事情。添加子工程和依赖后，Xcode 知道去哪里寻找头文件和二进制文件，也知道根据你的设置去选择哪需要构建哪一个架构的库。这非常方便。

如果你使用你自己的库或者拥有源代码和工程文件，将库作为子工程不失为一个引入静态库的简便的方法。让你更容易作为工程依赖构建整合，并担心更少的事情。

未来

你可以从这里下载到包括了本教程所有代码的工程。

希望本教程能够让你对静态库的基本概念和怎样在应用中使用它们有一个更深入的了解。

下一步就是用所学到的知识构建你自己的库了。你肯定有一些添加到工程中通用类。它们是你加入你自己的可复用库的优秀候选人。你也可以考虑根据功能创建多个库:网络部分的代码作为一个，UI 部分作为另一个，等等。你可以只向工程中添加所需要的代码。

为了强化和深入探讨你在本教程中所学到的概念，我推荐苹果的文档 Introduction to Using Static Libraries in iOS。
原文

http://www.cocoachina.com/applenews/devnews/2013/1204/7468.html?utm_source=Tuicool_Weekly

开始 iOS 7 中自动布局教程(一)

到目前为止，如果你的设计相当的复杂，那么你必须编写大量的代码来适应这样的布局。你应该很高兴，现在这样的情况再也不存在了--iOS6 为 iPhone 和 iPad 带来了一个极好的新特性：自动布局。X

提示：团队成员 Jatthijs Hollemans（iOS 初级系列作者）已经将这篇文章移植到 iOS7 feast 上。希望你能够喜欢。

你是否曾经想让你的 app 在横竖屏方向上看起来都表现良好而受

挫？是否在做支持 iPhone 和 iPad 屏幕布局界面时几近大小便失禁？今天我将给你带来好消息！

一直为大小相同的屏幕设计一个用户界面并不难，但如果屏幕的尺寸改变的话，UI 元素的位置和大小也需要相应的做出改变。

到目前为止，如果你的设计相当的复杂，那么你必须编写大量的代码来适应这样的布局。你应该很高兴，现在这样的情况再也不存在了--iOS6 为 iPhone 和 iPad 带来了一个极好的新特性：自动布局。Xcode 5 和 iOS7 中对自动布局做出了改善！如果你曾经尝试着在 Xcode4 中使用自动布局并最终做出放弃，现在是该给 Xcode5 一次机会了。

在你的程序中，自动布局不仅可以很容易的支持不同大小的屏幕，一个额外的功能，它也使得本地化几乎变得微不足道。你不再需要为每种你希望支持的语言创建新的 nibs 或 storyboards，包括像 Hebrew 或 Arabic 这样从右到左的语言。

这个教程将向你展示如何使用 Interface Builder 开始自动布局.在 iOS6 教程中，我们进一步讨论过这个教程，然后有一个基于这个知识点完整的新章节，并且向你展示如何通过代码完全释放出自动布局的能量。

注意：我们已经开始将 iOS6 教程中相应的章节更新到 iOS7-这只是

先给大家解解馋！当我们完成后，所有 iOS6 PDF 教程的订阅者将会得到免费的更新下载。

so，备好零食和你喜欢的咖啡，准备成为自动布局的达人吧！

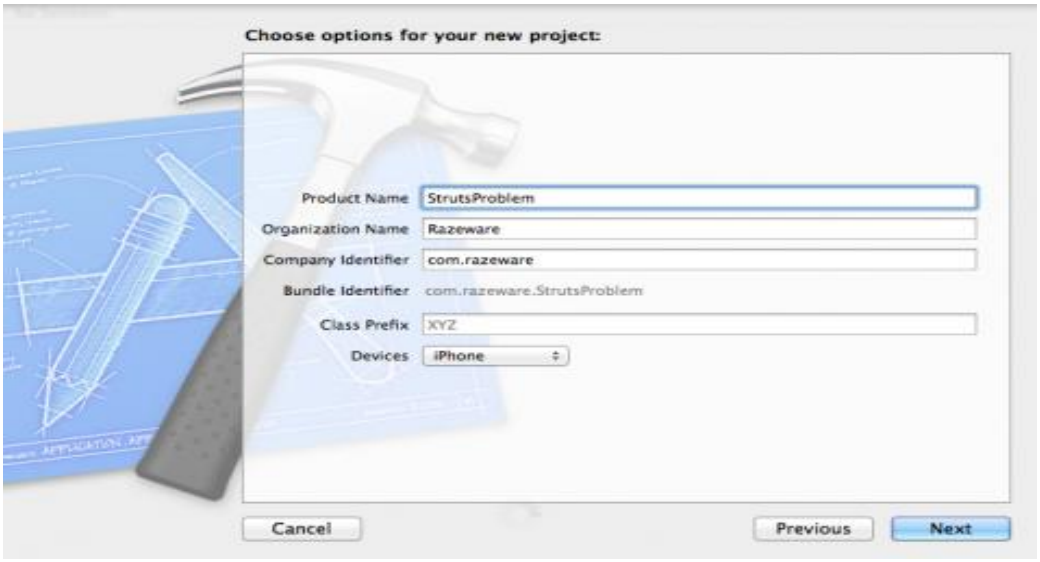
springs 和 struts 的问题

你肯定很熟悉 autosizing masks-也被认为是 springs&struts 模式。autosizing mask 决定了当一个视图的父视图大小改变时，其自身需要做出什么改变。它有一个灵活的或固定不变的 margins（struts）吗？它的宽和高要做出什么改变（springs）？

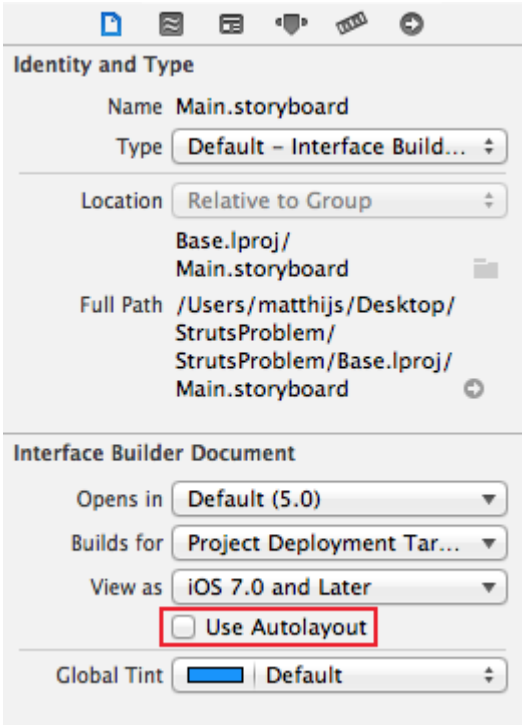
举个例子，一个宽度灵活的视图，如果其父视图边框，那么它也会相应的变宽。一个视图右边拥有固定的 margin，那么它的右边缘将会一直粘住其父视图的右边缘。

autosizing 系统在简单的情况下非常奏效，但当你布局变得更复杂时，它立马跪了。让我们看一个 springs 和 struts 不能处理的示例。

打开 Xcode5，创建一个基于 Single View Application 模板的 iPhone 项目。叫做"StrutsProblem":



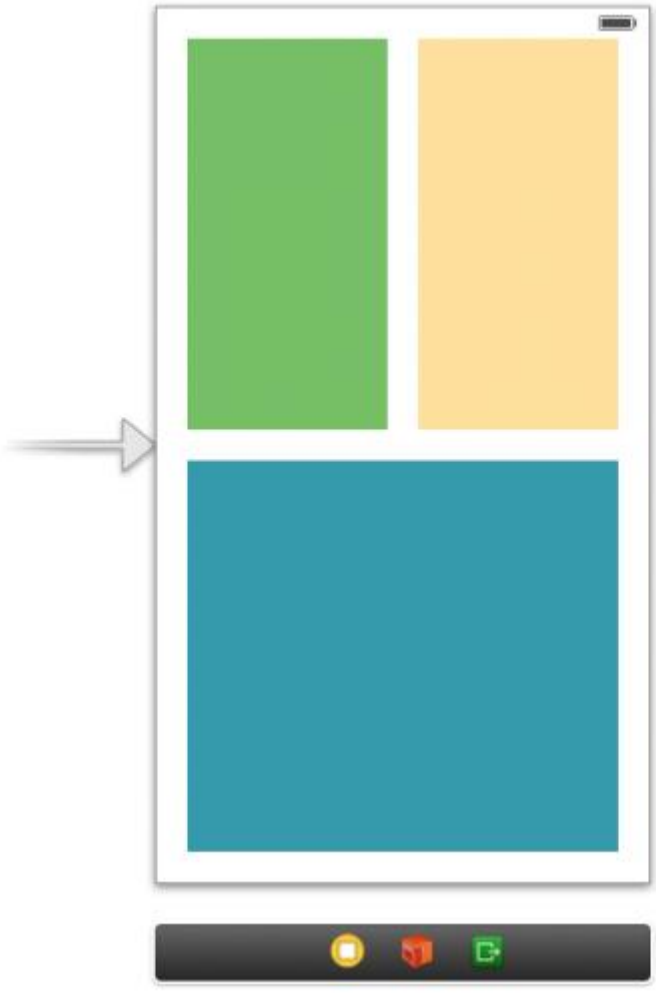
点击 Main.storyboard。在你做别的之前，首先将这个 storyboard 的自动布局关了。你需要在 File inspector，第一个选项的第六个 tabs 里：



将 Use Autolayout 的 box 勾选去掉。现在 storyboard 使用旧的 struts-and-springs 模型。

注意:任何你使用 Xcode4.5 或更高版本中，nib 或者 storyboard 文件都默认激活了自动布局。因为自动布局是 iOS6 以及以上系统的一个新特性，如果你想使用最新的 Xcode 开发兼容 iOS5 的程序，你需要将这个选项去掉。

拖拽三个新的视图到主视图上，并且像这样排列起来：



为了表述更清楚，这里给出每个视图的颜色，这样你就能分清哪个是驴子哪个是马了。

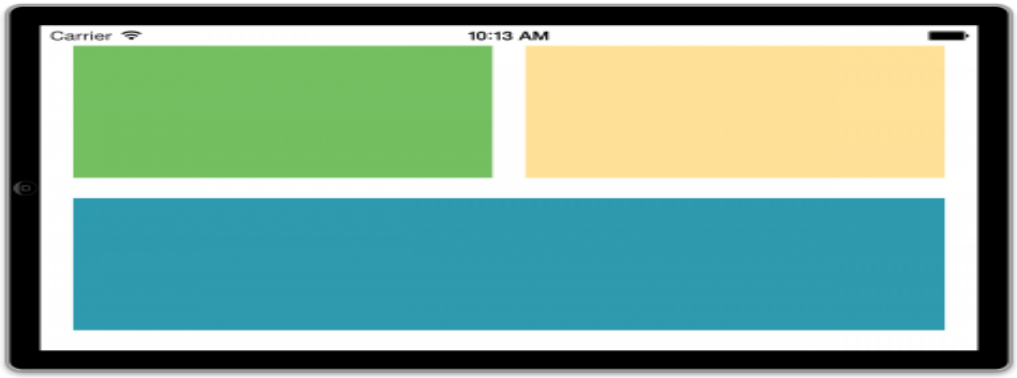
每个视图的 inset 到窗口的距离都是 20 点；视图之间的距离也是 20

点。底部的视图的宽是 280 点，上面两个视图的宽都是 130 点。所有的视图的高都是 254。

在 iPhone Retina 4-inch simulator 上运行这个程序，并且将模拟器旋转到 landscape。程序看起来便变成这副鬼样，这不是我想象的那样：

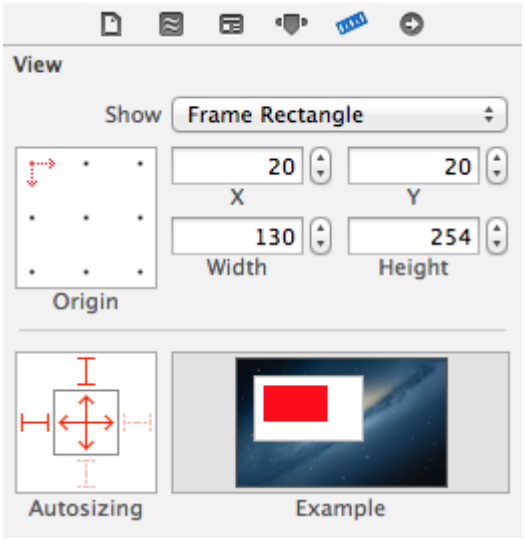
注意:你可以使用 Hardware\Rotate Left 和 Rotate Right 菜单选项旋转模拟器，或者通过按下键盘上的? 键，同时按下←或→。

而你想象的程序在 landscape 应该像这样：



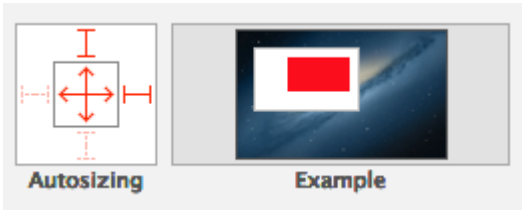
很明显，三个视图的 autosizing masks 留下了一些需要改进的地方。

将左上方视图的 autosizing 设置改成这样：

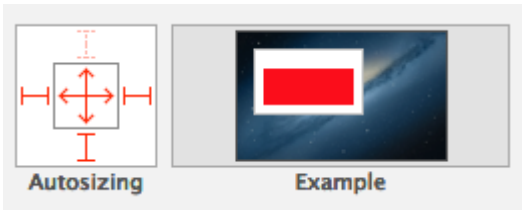


这将会让视图贴附左上边缘（不是右下边缘），并且当父视图大小改变时，重新调整自身水平和垂直方向的大小。

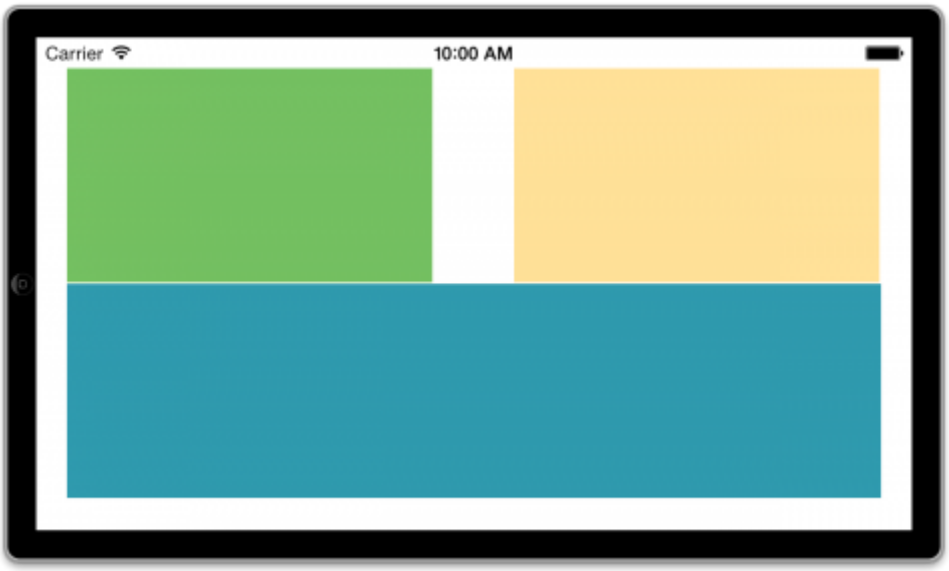
同样的，右上方视图的 autosizing 设置改成这样：



底部视图：



再次运行程序，并且旋转到 landscape。现在看起来像这样：



已经很接近了，但又不完全一样。视图之间的 padding 不正确。换个说法就是视图的大小不完全正确。问题出在当父视图改变大小时，autosizing masks 告诉子视图调整大小，但又没告诉子视图该调整多少（坑儿？）。

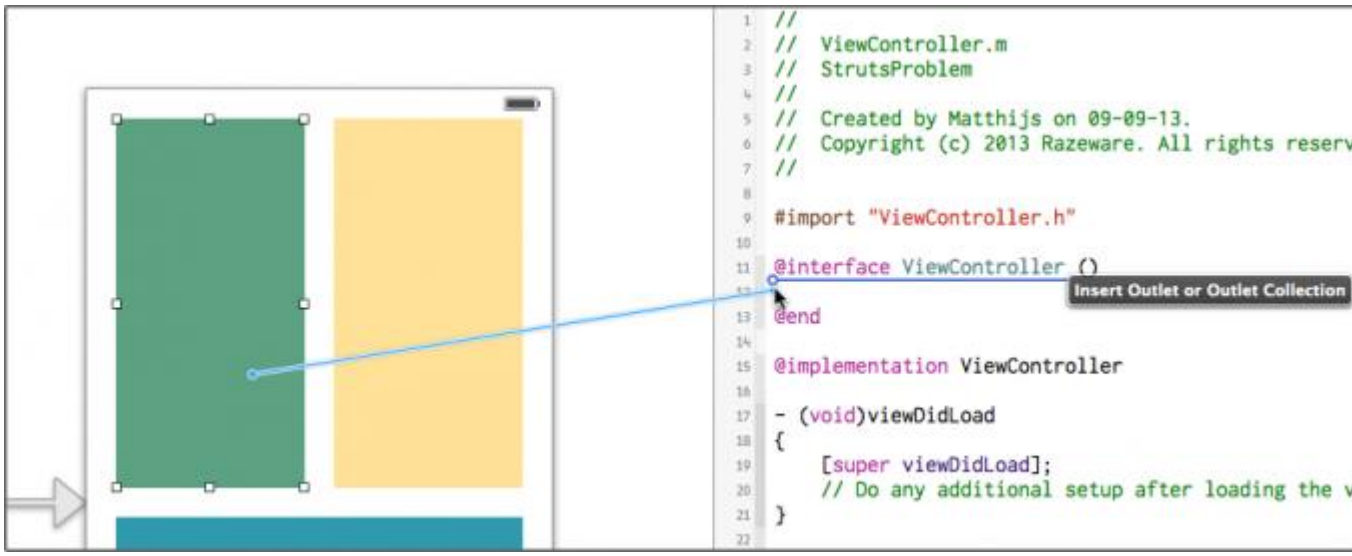
你可以调戏 autosizing masks-比如，改变灵活宽度和高度设置（springs）-你不会得到完全正确的三个间距 20 点的视图。

为了解决这个 springs 和 struts 方法的布局问题，非常不幸，你需要额外写一些代码。

在旋转用户界面之前、之间、之后，UIKit 会发送一些消息到你的视图控制器，你可以截获这些消息，从而对你 UI 做出改变。代表性的像 viewWillLayoutSubviews，你会重写这个方法从而改变任何需要重新排列的视图的 frame。

在这之前，你需要先做出一个 outlet 属性来引用这个视图。

切换到 Assistant Editor 模式，按住 Ctrl，将三个视图都拖到 ViewController.m 中去：



分别链接视图到这三个属性：

```
1. @property (weak, nonatomic) IBOutlet UIView *to
   pLeftView;
```

```
2. @property (weak, nonatomic) IBOutlet UIView *topRightView;
3. @property (weak, nonatomic) IBOutlet UIView *bottomView;
```

下面的代码写到 ViewController.m:

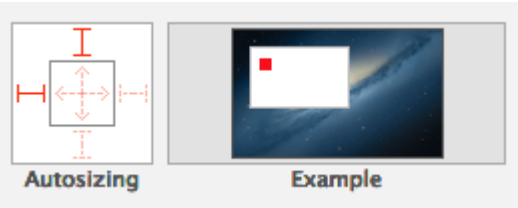
```
1. - (void)viewWillLayoutSubviews
2. {
3.         if (UIInterfaceOrientationIsLandscape(self.interfaceOrientation))
4.         {
5.                 CGRect rect = self.topLeftView.frame;
6.                 rect.size.width = 254;
7.                 rect.size.height = 130;
8.                 self.topLeftView.frame = rect;
9.
10.                rect = self.topRightView.frame;
11.                rect.origin.x = 294;
12.                rect.size.width = 254;
13.                rect.size.height = 130;
```

```
14.                 self.topRightView.frame  =  rect;
15.
16.                 rect  =  self.bottomView.frame;
17.                 rect.origin.y  =  170;
18.                 rect.size.width  =  528;
19.                 rect.size.height  =  130;
20.                 self.bottomView.frame  =  rect;
21.         }
22.         else
23.         {
24.                 CGRect  rect  =  self.topLeftView.fr
    ame;
25.                 rect.size.width  =  130;
26.                 rect.size.height  =  254;
27.                 self.topLeftView.frame  =  rect;
28.
29.                 rect  =  self.topRightView.frame;
30.                 rect.origin.x  =  170;
31.                 rect.size.width  =  130;
32.                 rect.size.height  =  254;
33.                 self.topRightView.frame  =  rect;
34.
```

```
35.                 rect  =  self.bottomView.frame;
36.                 rect.origin.y  =  295;
37.                 rect.size.width  =  280;
38.                 rect.size.height  =  254;
39.                 self.bottomView.frame  =  rect;
40.         }
41. }
```

当视图控制器旋转到一个新的方向，这个回调将会被调用。它会监控视图控制器旋转的方向，并且适当的调整视图大小-在这种情况，根据已知 iPhone 屏幕大小会有一个 hard-code(将可变变量用一个固定值来代替的方法叫做 hard-code)偏移。这个回调会在一个动画 block 中发生，所以会动态的改变大小。

暂时还不要运行这个程序。首先你需要按下面的样子重新保存三个视图的 autosizing masks，否则 autosizing mechanism 将会和你在 viewWillLayoutSubviews 中设置的位置和大小冲突。



这样就可以了，运行程序并且翻转到 landscape。现在视图排列的非常号。翻转回到 portrait，经核实，一切都良好。

这样奏效了，但是你需要为这个非常简单的例子编写大量的布局代码。想象一下，为布局付出的努力是非常复杂的，特别是个别视图动态的改变大小，或者子视图的个数是不固定的。

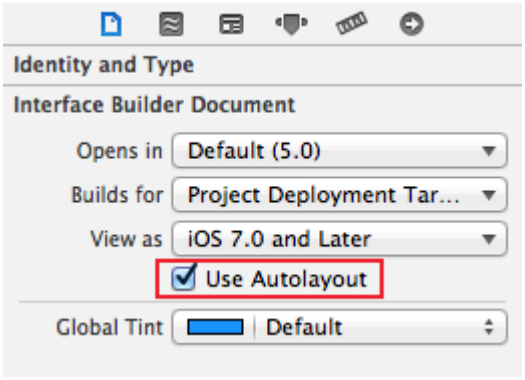
现在试着在 3.5 寸的模拟器上运行程序。我了个去。视图的位置和大小又错了，因为 viewWillLayoutSubviews 的 hard-code 坐标是基于 4 英寸大小的手机(320x568 取代 320x480)。你可以增加另一个 if 语句判断屏幕大小，并使用不同的坐标集，但是你可以看到这个方法很快变得不切实际。

注意:另一个你可以采取的方法就是为 portrait 和 landscape 模式建立独立的 nibs。当设备旋转时，你从另一个 nib 中装载视图并替换掉当前的那个。但这任然需要做很多工作，并且维护两个 nibs 也会增加问题。当你使用 storyboards 替代 nibs 的时候，这个方法也变得不切实际。

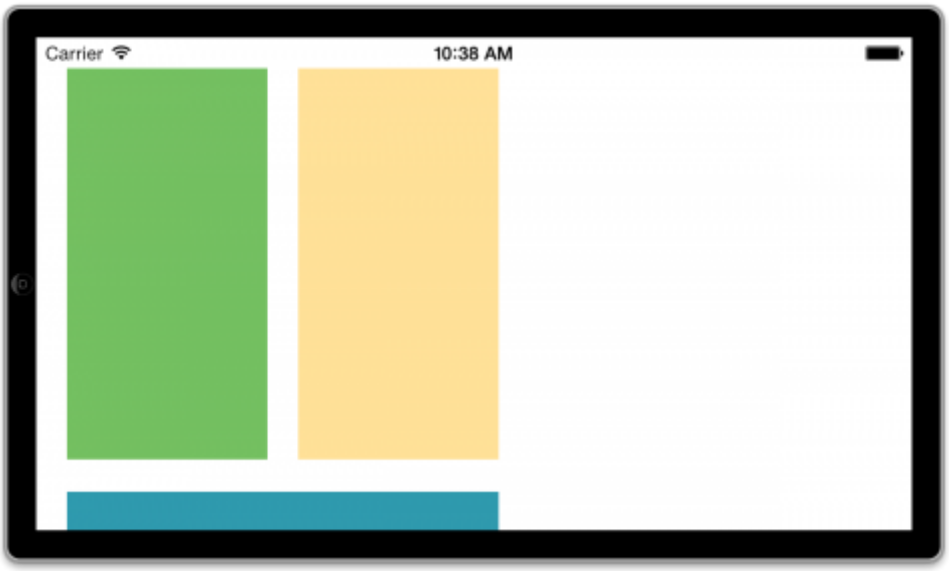
自动布局拯救猿！

现在你将会看到如何用自动布局实现相同的效果，从 ViewControlle r.m 中移除 viewWillLayoutSubviews，因为我们不再需要写任何代码。

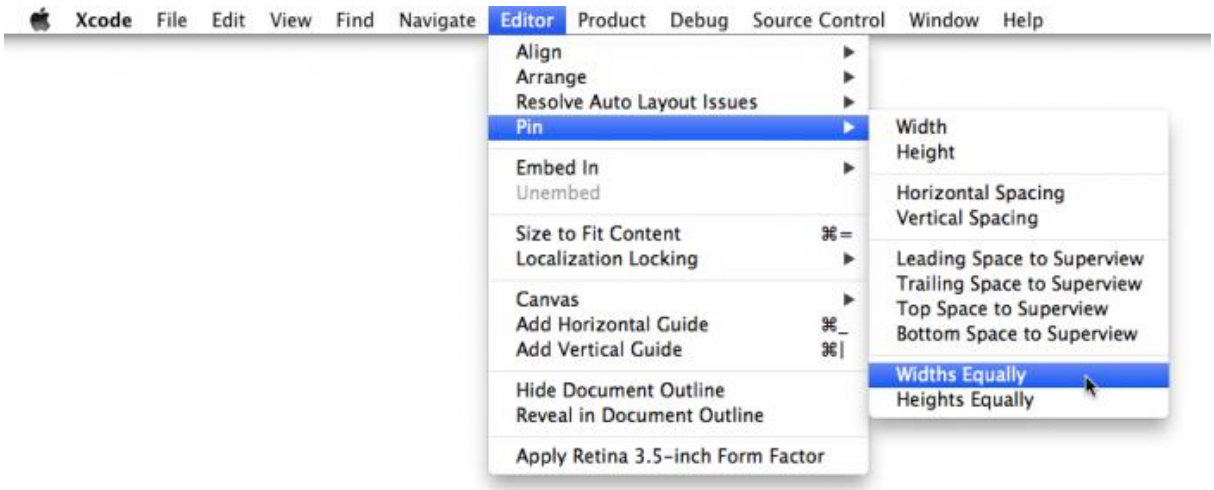
选择 Main.storyboard，并在 File inspector 中选择开启 Use Auto layout：



运行程序，旋转到 landscape。现在看起来像这样：

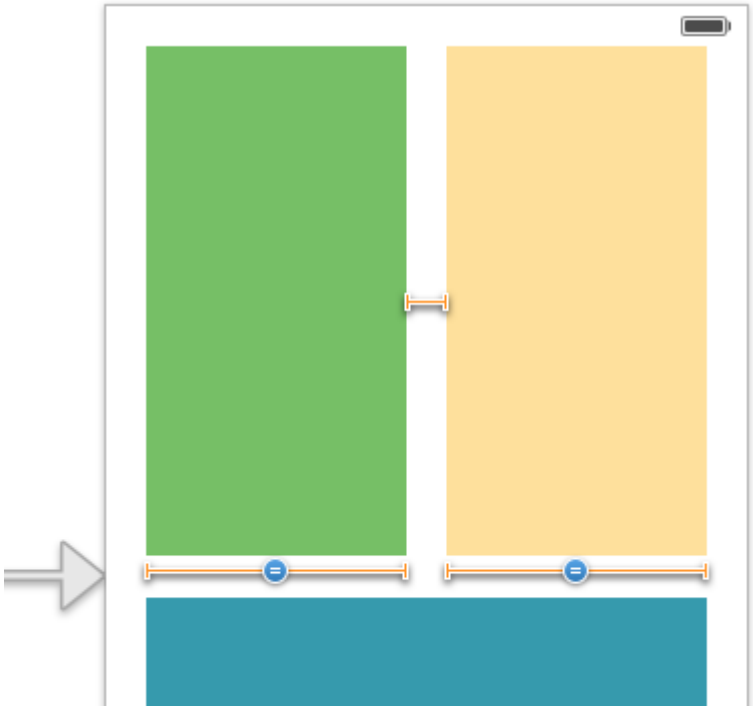


让我们把自动布局付诸行动。当你点击顶部两个视图时，按住?键，这样两个视图都被选中了。从 Xcode 的 Editor 菜单中选择 Pin\Widt hs Equally:



再次选中两个相同的视图，选择 Editor\Pin\Horizontal Spacing。（尽管你执行完第一次 Pin 处理后，两个视图看起来还是被选中的，但其实他们只是在一个特别的布局关系显示模型里。所以你需要重新选择这两个视图）

storyboard 现在看起来像这样：



橙色的"T-bar"形状代表视图间的约束。目前为止你增加了两个约束：一个等宽约束和一个位于两个视图间的水平约束。约束表达了视图之

间的关系，并且他们是你使用自动布局建立布局最主要的工具。这货看起来有点吓人，但是一旦弄懂它的意思，便变得相当简单。

为了继续为这个屏幕简历布局，执行下面这些步骤。每个步骤增加更多橘黄色的 T-bars.

- Top Space to Superview
- Leading Space to Superview

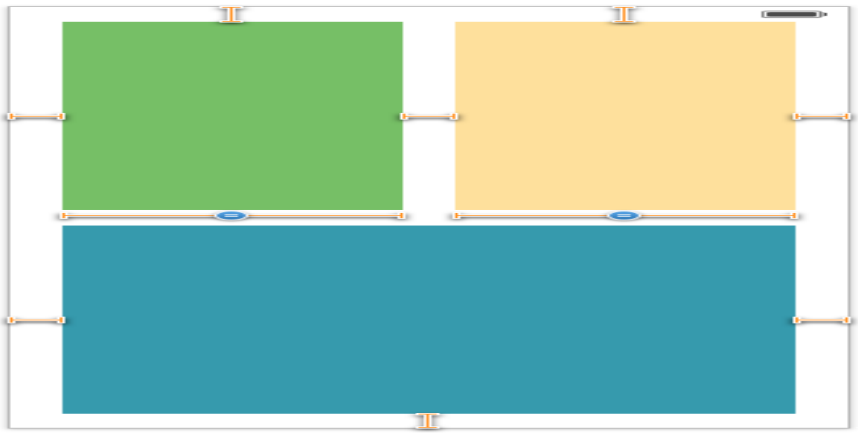
For the view on the right, choose:

- Top Space to Superview
- Trailing Space to Superview

And for the big view at the bottom:

- Leading Space to Superview
- Trailing Space to Superview
- Bottom Space to Superview

现在你应该有了下面这些约束：



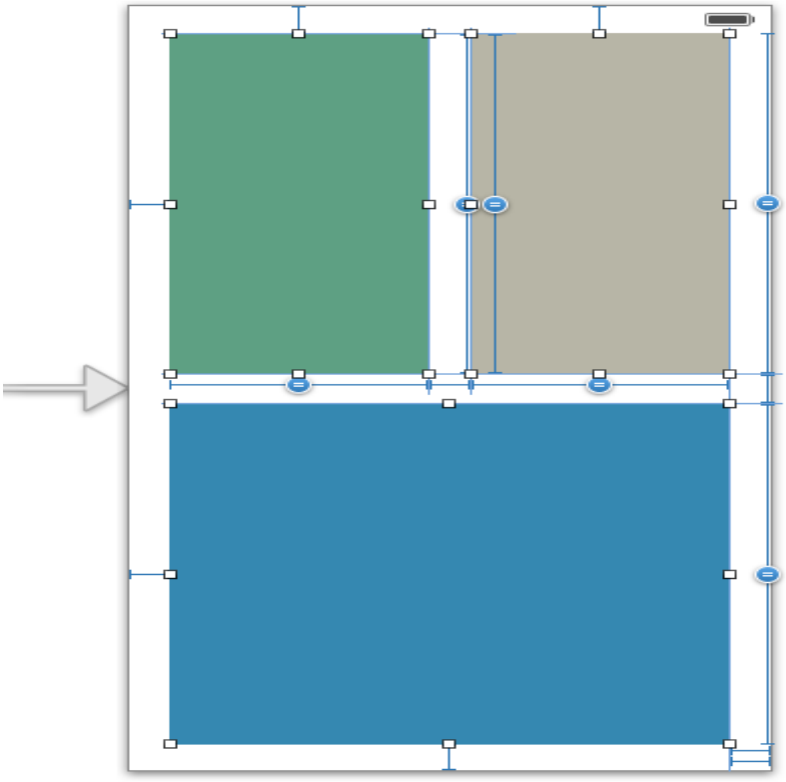
注意 T-bars 仍然是橘黄色的。这意味这你的布局没有完成；自动布

局没有足够的约束条件计算出视图的位置和大小。解决办法便是增加更多约束，直到他们变蓝。

按下? 键并选中三个视图。从 Editor 菜单中，选择 Pin\Heights Equally。

现在选中左上角的视图和底部视图（像前面一样按住? 键），选择 Editor\Pin\Vertical Spacing.

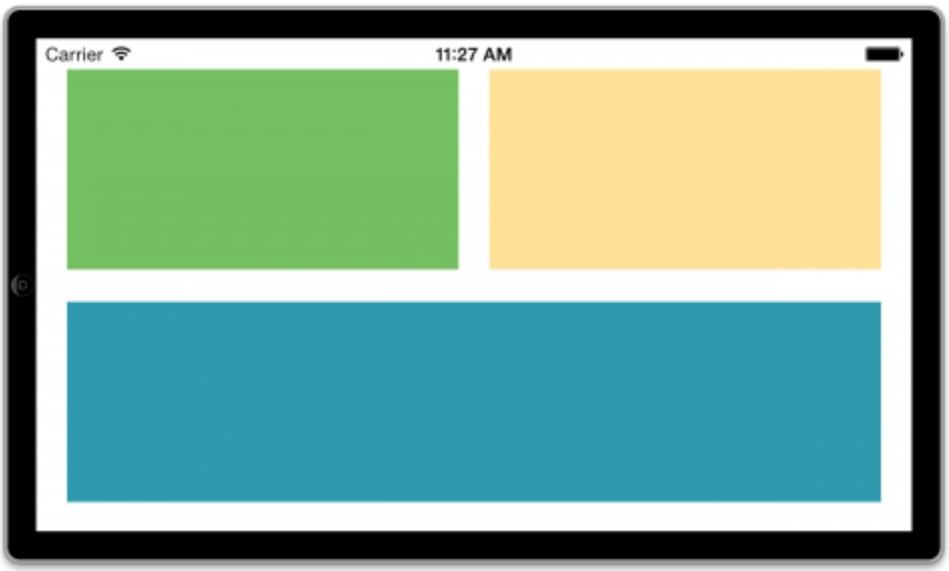
Interface Builder 看起来应该像这样：



T-bars 已经变蓝了。自动布局现在已经有足够的信息来计算出一个有效的布局。这看起来有点杂乱无章，这是因为等宽和等高约束条件

占去了很大空间。

运行程序并且...我说吧，不需要写一行代码便运行的很好了。不管你在哪个模拟器上运行；在 3.5 英寸和 4 英寸设备上，布局都运行良好。



这非常酷，但是究竟你在这做了什么？自动布局让你表达出布局中的视图和其他每个视图的关系，而不是需要你指出视图有多大，放在哪儿。你需要放置以下这些关系（即我们所谓的约束）到布局中：

1.左上角和右上角的视图总是有相等的宽度（也就是 pin 中第一个 widths equally 命令）。

2.左上角和右上角的视图水平方向有 20 点距离（也就是 pin 中的 horizontal spacing）。

3.所有的视图总是有相同的高度（也就是 pin 中 heights equally 命令）。

4.上面两个视图和下面那个视图垂直方向上有 20 点距离（也就是 pi

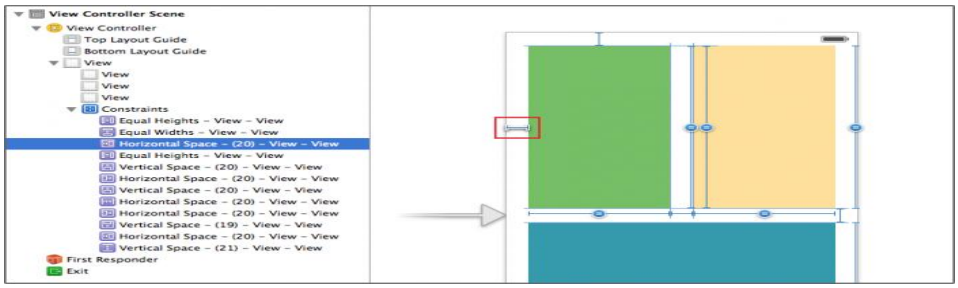
n 中的 vertical spacing)。

5.视图和屏幕边缘有 20 点空间（top，bottom，leading 和 trailing space 相对于父视图的约束)。

这些就足以表达出自动布局该怎么放置视图，以及当屏幕大小改变时该如何处理。

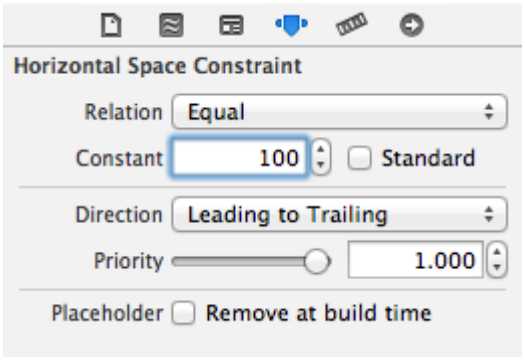
你可以在左边 Document Outline 中看到你所有的约束，组名叫做 Constraints（当你为 storyboard 激活自动布局时才会加进来)。

如果你在 Document Outline 中点击一个约束，Interface Builder 将会在视图中高亮出它：

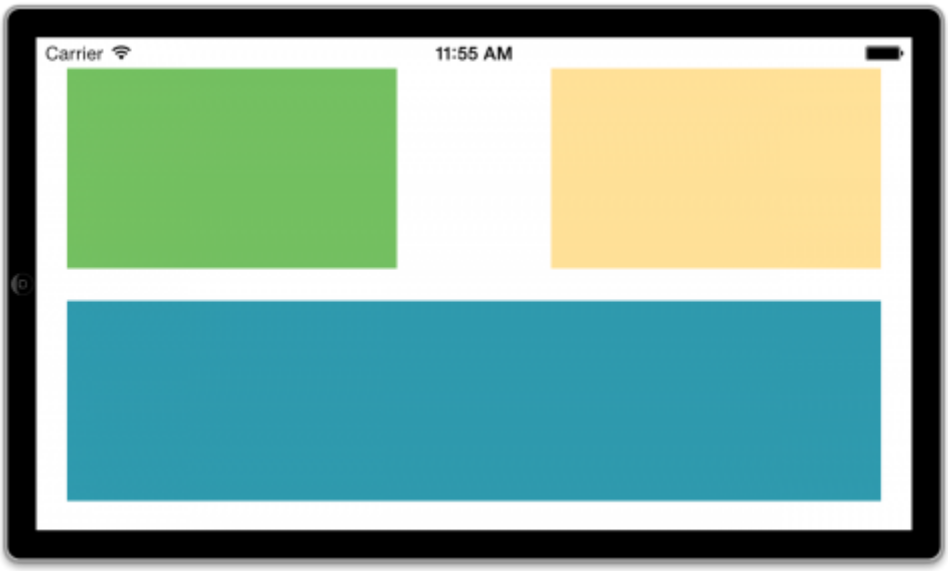


约束是一个真实的对象（NSLayoutConstraint)，并且他们也有属性。

比如，选择上面两个视图的间距约束条件（叫做"Horizontal Space（20）"），然后切换到 Attributes inspector。你可以在那里通过编辑 Constant 字段改变边缘空间的大小。



将它设置为 100，然后再次运行程序。现在他们边缘空间变得更宽了：



自动布局在描述视图上比 springs 和 struts 显得更有表现力。在这篇教程的剩余部分，你将会学到约束的一切，以及如何将他们应用到 Interface Builder 上来构造出不同种类的布局。

自动布局如何工作

正如你在上面测试样例中所看到的一样，自动布局最基本的工具是约束。一个约束描述了两个视图间的几何关系。比如，你可能有这样一

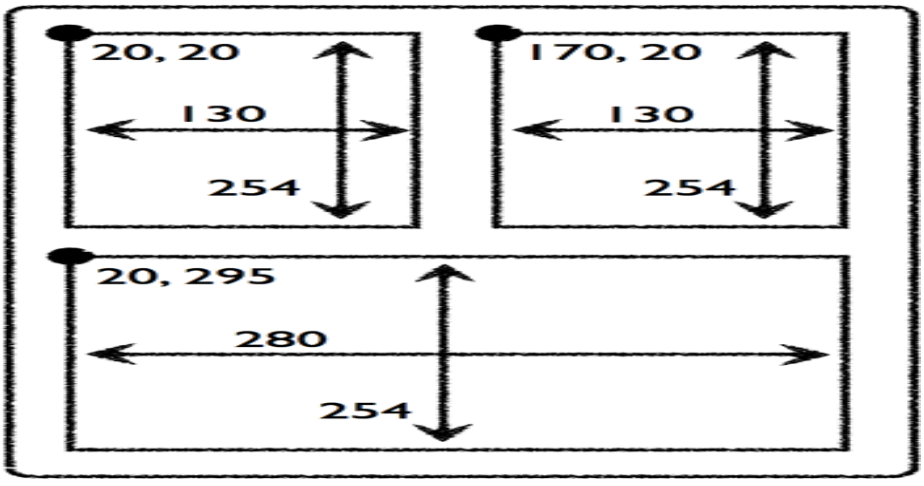
个约束：

"label A 右边缘和 button B 左边缘有 20 点的空白空间。"

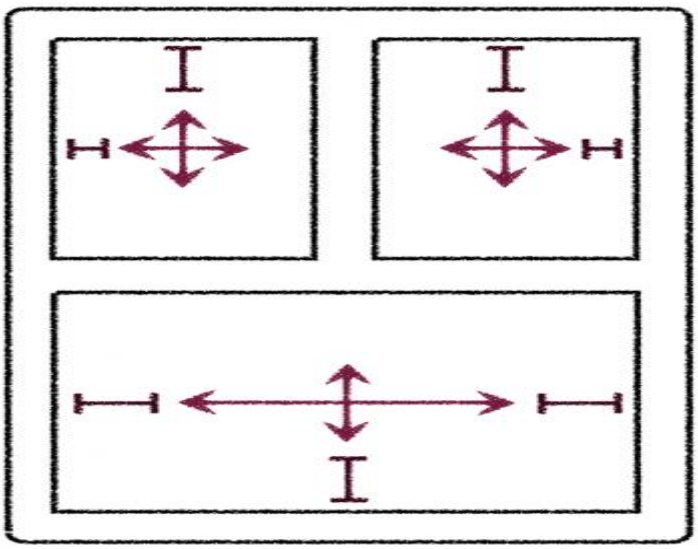
自动布局会考虑到所有的约束，然后为你的视图计算出理想的位置和大小。你再也不需要亲自为你的视图设置 frames 了-自动布局会完全基于你为这些视图设置的约束为你做这个工作。

自动布局以前，你一直需要为视图的 frames 设置 hard-code，要么在 Interface Builder 中将他们放置在特定的坐标，或通过传递一个 rectangle 到 initWithFrame:，或者设置视图的 frame，bounds 或者 center 属性。

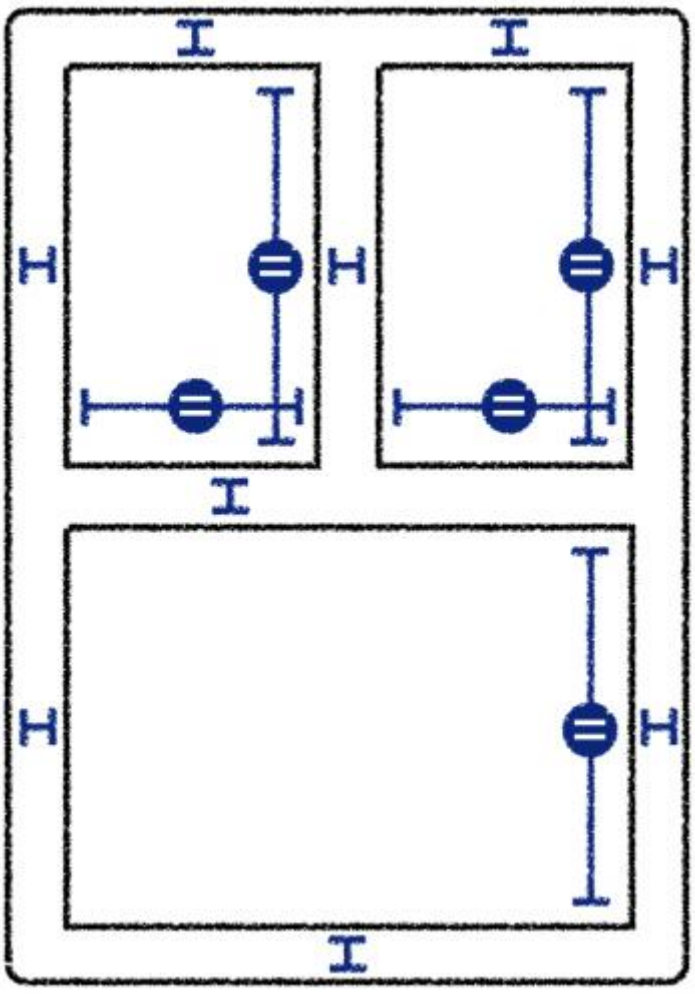
就你刚刚做的那个程序，你需要明确设置 frames 为：



还需要为这些视图设置自动调整大小的 masks：



这再也不是你需要为屏幕设计所考虑的东西了。使用自动布局，你需要做这些：



视图的大小和位置再也不重要了，只有约束要紧。当然，当你拖一个新建的 button 或 label 到画布上时，它会有一定的大小，并且你会将它拖到某一位置，但这是只一个用来告诉 Interface Builder 如何放置约束的设计工具。

想你所想，如你所愿

使用约束最大的优势就是你再也不需要把时间浪费在坐标上了。相反，你可以向自动布局描述视图如何和其他视图相关联，自动布局将会为你完成所有困难的工作。这叫做根据目的设计（designing by intent）。

当你根据目的设计时，你表达的是你想要实现什么，而不需要关心它如何实现。"button 的左上角坐标为（20，230）"，现在你可以这么说了：button 是垂直居中于它的父视图，并且相对于父视图的左边缘有一个固定的距离。

使用这个描述，不管父视图多大或多小，自动布局都可以自动计算出你的 button 需要在哪儿出现，

其他根据目的设计的示例（自动布局可以处理所有这些指令）：

"这两个 text fields 的大小需要一直相等。"

"这两个 button 需要一直一起移动。"

"这四个 labels 需要一直右对齐。"

这使得你用户界面的设置更具描述性。你只需简单的定义约束，系统会为你自动计算 frames。

在第一部分你看到，即使为几个视图在横竖方向上正确的布局都需要做大量的工作。有了自动布局，你可以绕过这些麻烦。如果你正确的设置了约束，那么在横竖屏方向上，布局将不需要做任何改变。

使用自动布局另一个重要的好处就是本地化。比如德语中的文本，出了名的比老奶奶的裹脚布还要长，适配起来是一件很麻烦的事。再次，自动布局拯救了猿，因为它能根据 label 需要显示的内容自动改变 label 的大小。

现在增加德语，法语或者其他任何一种语言，都只是设置约束的事，然后翻译文本，然后。。。就没有然后了！

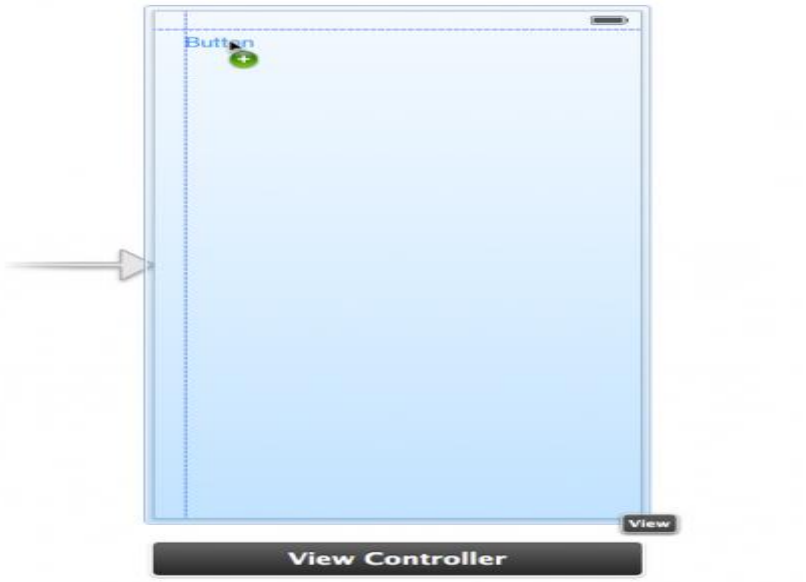
获得自动布局窍门最好的方法就是使用它，所以这正是剩下教程中你会学到的东西。

注意：自动布局不仅对旋转有作用；它还能轻易的缩放你 UI 的大小从而适应不同尺寸的屏幕。这并不是巧合，当 iPhone5 拥有更高屏幕的同时，这个技术也同时加到了 iOS 中！自动布局能轻易的拉伸你程序的用户界面，从而充满 iPhone5 垂直方向上多出来的空间。随着 iOS7 中的动态类型，自动布局变得更加重要了。用户现在可以改变全局字体大小设置--有了自动布局，这将变得非常简单。

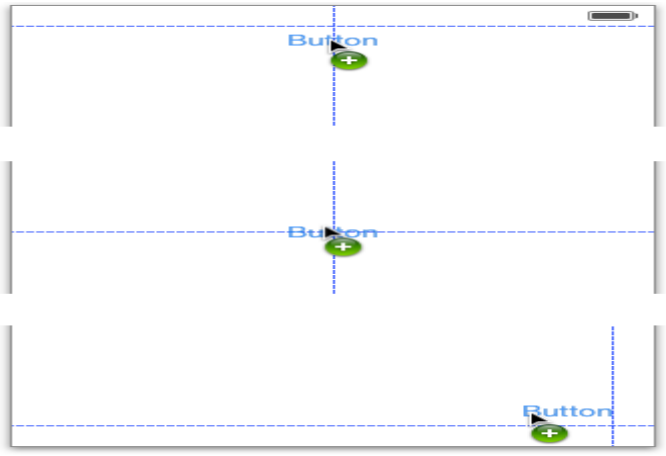
拥抱约束（courting constraints）

关闭你当前的项目并用 Single View Application 模板创建一个新的 iPhone 项目。叫做"Constraints"。任何用 Xcode5 创建出来的项目都会自动假定你会使用自动布局，所以你并不需要额外做任何事情。

点击 Main.storyboard 打开 Interface Builder。拖一个新的 Button 到画布上。注意当你拖拽的时候，蓝色虚线将会出现。这写线用来做向导。

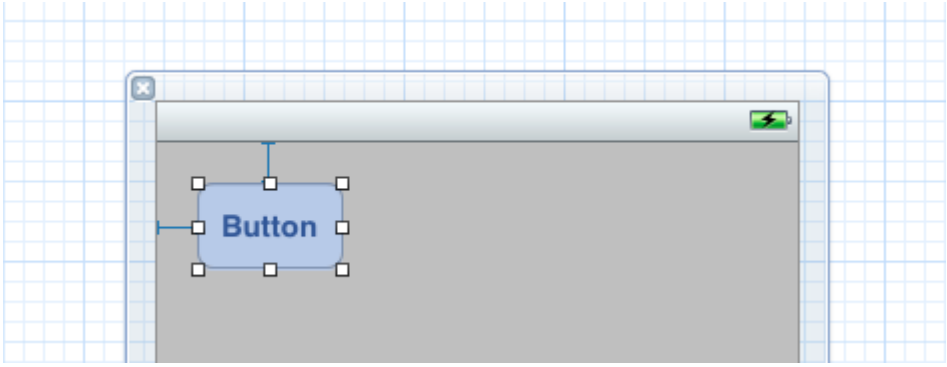


在屏幕边缘以及中心的时候，都会有向导线：



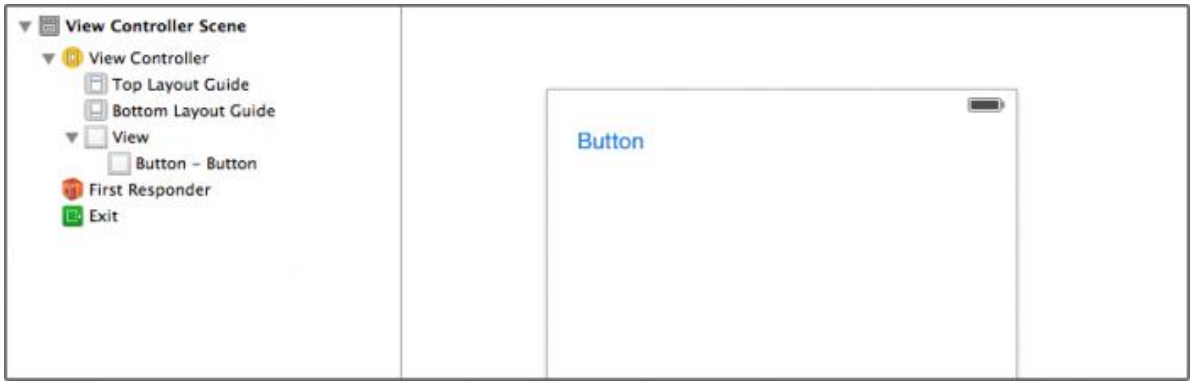
如果之前你已经使用过 Interface Builder，那么你肯定看到过这些向导线。这对我们对齐控件有很大的帮助。

在 Xcode4 中激活自动布局时，向导线有另外一个目的。你任然可以用他们来对齐，但是他们也会告诉你新的约束将会在哪儿。如果你将 button 沿着向导线反方向拖拽到左上角时，Xcode4 中的 storyboard 看起来便像这样：



有两个蓝色的东西附属在 button 上面。这些 T-bar 形状的对象便是约束了。Xcode 4 的 Interface Builder 中不管你将 UI 控制器放在哪儿，它总是会给出有效的约束。理论上这听起来是个好主意，但是实践起来，在 Interface Builder 中使用自动布局却非常困难。

幸运的是，Xcode5 中已经有所好转。将 button 拖拽到画布上之后并看不到 T-bars 形状的东西：



同时在 Document Outline 面板中也没用 Constraints 部分。得到结

论：此时 button 上并没有设置任何约束。

那这是如何运作的呢？我们之前了解的自动布局总是需要足够多的约束才能决定视图的大小和位置，但是现在我们这儿跟本没有约束。确定这是一个完整的布局？

这这是 Xcode5 相对 Xcode4 来说最大的一个提升：再也不强制你总是有一个有效的布局。

注意：1.运行一个无效布局的程序是不明智的，因为自动布局不能正确的计算需要将视图放在哪儿。要么视图的位置是不可预知的（约束不够），要么程序将会崩溃（约束过多）。

2.Xcode4 设法保证总是有足够多正确的约束来创建一个有效的布局。不幸的是，它经常会将你的约束替换为你并不想要的。这会令人很沮丧，正是因为这个原因很多开发者放弃了自动布局。

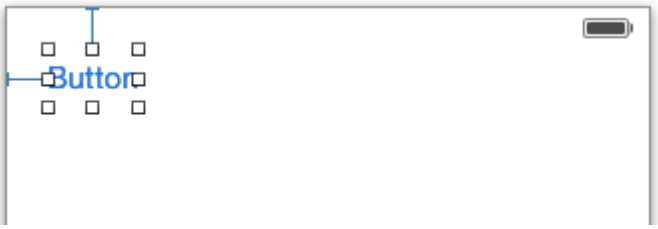
3.Xcode5 中，当你编辑 Storyboard 时它允许你有不完整的布局，但它也会指出哪些地方你还需要修改。使用 Interface Builder 创建的自动布局驱动用户界面变得更有趣了，使用 Xcode5 也消耗更少的时间。

如果你根本不提供任何约束，Xcode 自动分配一套默认的约束，正是我们所知的自动约束。它会在程序 built 的编译时间中去完成这些事，而不是设计时间。当你设计你的用户界面时，Xcode5 中的自动布局为了不参与你的设计方法而努力工作，这这是我们喜欢它的原因。

自动约束为你的视图提供一个固定尺寸和位置。换句话说，视图总是拥有跟你在 storyboard 中看到的一样的坐标。这是非常方便的，因为这就意味着你可以大量的忽视自动布局。你可以为那些拥有充分约束的控件不增加约束，只为那些需要特殊规则的视图创建约束。

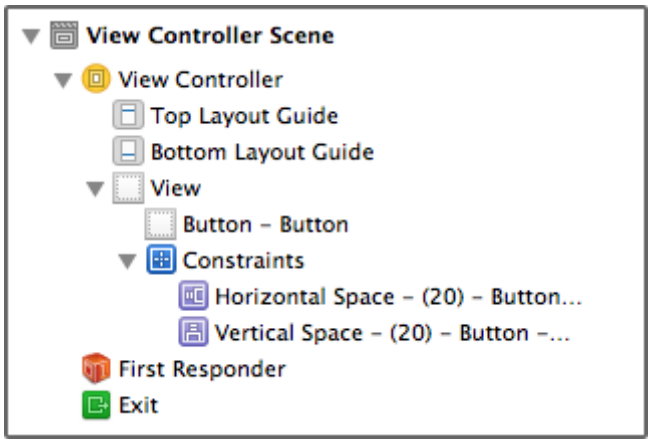
OK，让我们玩一玩约束并看看他们能做什么。现在，按钮是在左上角，并且没有约束。确认按钮跟两个拐角向导线对齐。

使用 Editor\Pin 菜单为按钮增加两个新的约束，看起来像这样：



这是 Leading Space to Superview 和 Top Space to Superview 选项。

所有的约束都会在 Document Outline 面板中列出来：



目前有两个约束，一个是 button 和 main view 左边缘的 Horizontal Space 约束，一个是 button 和 main view 上边缘的 Vertical Space 约束。这个关系通过约束描述起来便是："button 总是位于其父视图左上角 20 点处。"

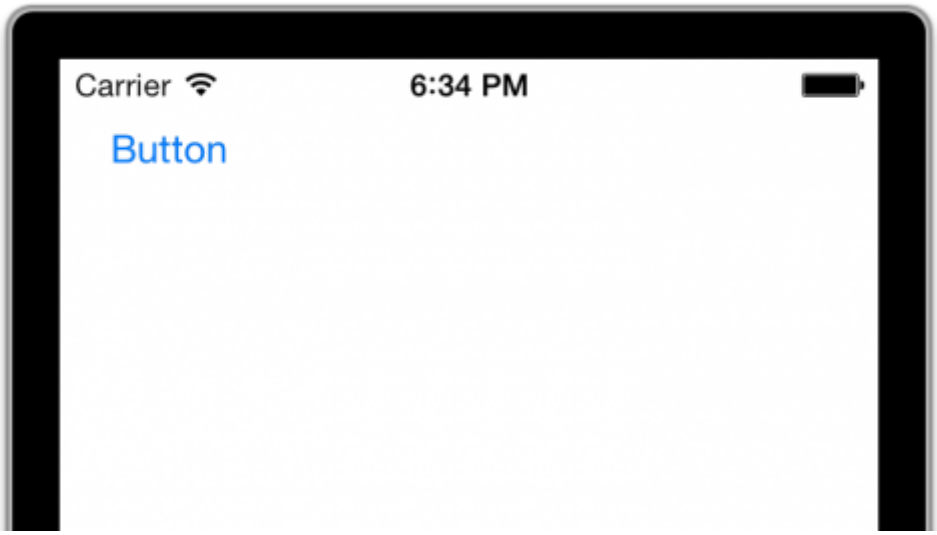
注意：这些其实都不是非常有用的约束，因为他们有相同的自动约束。如果你总是想你的 button 相对于父视图左上角，那么你还不如不提供任何约束，让 Xcode 为你做这些。

现在拖动 button 并将它放到屏幕的右上角，再次和蓝色向导线对齐：

哇哦，这里发生了什么？在 Xcode4 中这会破坏旧的约束并赋值一个基于蓝色向导线的新约束，但是在 Xcode5 中 button 保留了现存的约束。但问题是 button 在 Interface Builder 中的大小和位置再也不和自动布局希望基于约束的大小和位置相符合了。这叫做错位的视

图。(misplaced view)

运行程序。Button 仍然会出现在屏幕的左上角：



当谈到自动布局，橙色代表坏的。Interface Builder 绘制两个橙色方块：一个是虚线边框，一个是实线边框。虚线方块是根据自动布局显示视图的 frame。实线方块是根据你在屏幕上放置的视图的 frame。这两个应该吻合的，但是这里并没有。

如何修改取决于你想要达到什么目的：

1.你想让 button 附属于屏幕左边缘 254 点处吗？在这种情况下你需要将现存的 Horizontal Space 约束变大 234 点。这正是橙色 badge 中"+234"的意思。

2.你想让 button 附属于屏幕的右边缘？那么你需要移除现有的约束并重新创建一个新的。

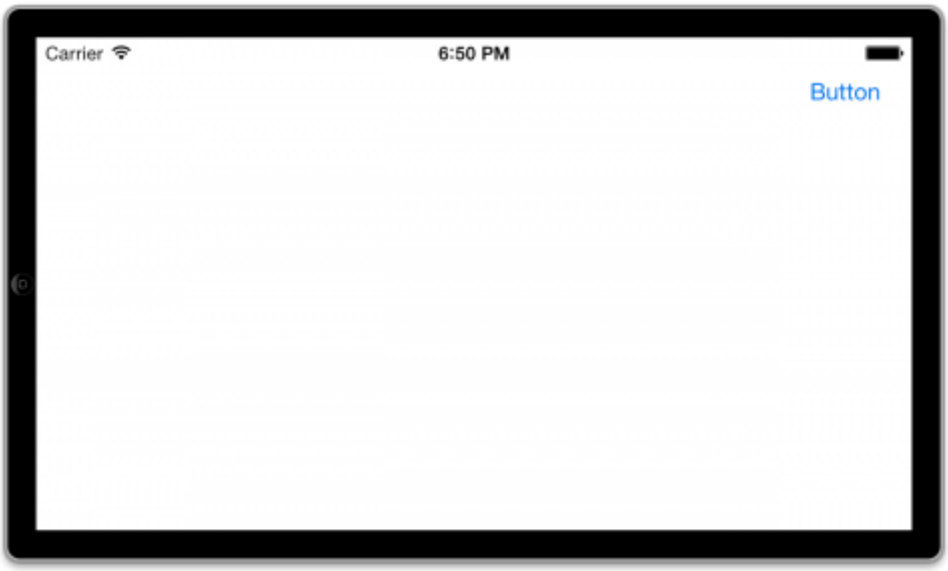
删除 Horizontal Space 约束。首先在画布或 Document Outline 中选中，然后按键盘上的 Delete 键。

注意这次 Vertical Space 约束变橙色了。直到现在它都是蓝色的。那一个约束并没有任何错误；它的意思是剩下的没有足够的约束决定 button 完整的位置。你任然需要在 X 轴方向增加一个约束。

Note：你可能会奇怪，为什么 Xcode 不为 X 轴方向自动增加一个约束。Xcode 中的规则是：Xcode 只为那些你没有设置任何约束的对象创建自动约束。一旦你增加一个约束，你便是告诉 Xcode 你接管了这个视图。Xcode 将不再增加任何自动约束，并希望你为这个视图增加需要的约束。

选中 button，并选择 Editor\Pin|Trailing Space to Superview. 这迫使在 button 右边缘和屏幕右边缘增加一个新的约束。关系表达如下："button 总是位于距离其父视图右上角 20 点处。"

运行程序并旋转到 landscape。注意 button 如何与屏幕右边缘保持相同距离：



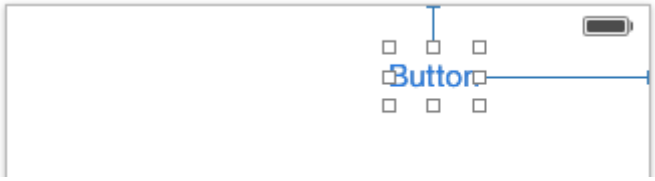
当你放置一个对立于向导线的 button（或者任何其他视图）并新建一个约束时，你会得到一个根据"HIG"（Apple's iOS Human Interface Guidelines document）定义的标准大小的间隔约束。对于边框来说，标准大小空间是 20 点。

现在将 button 向左拖拽一点：

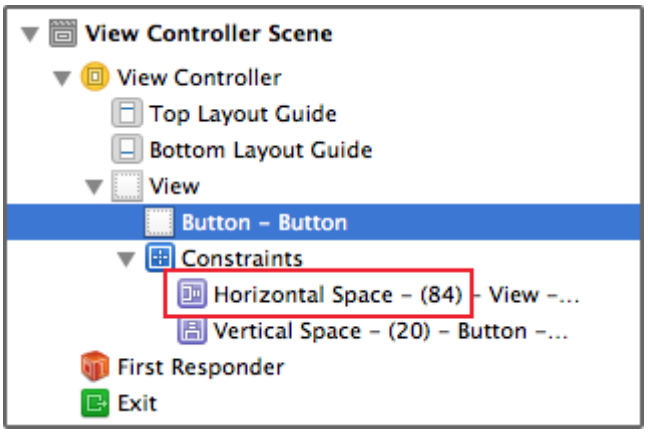
由于视图错位，你得到了一个橙色虚线边框。我们假设这个 button 新位置的确是你想要的。创建完一个约束后做一些细微的调整是很常见的，但这却会导致橙色方块出现。一个修改方法就是移除约束并创建一个新的，但还有一个更简单的解决方案。

Editor 菜单上有一个 Resolve Auto Layout Issues 子菜单。从这个

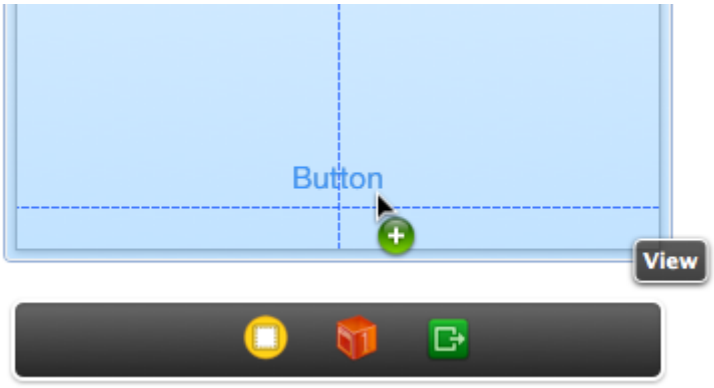
菜单中，选中 Update Constraints。就我这个情况来说，这会告诉 Interface Builder 需要将约束变大 64 点，像这样：



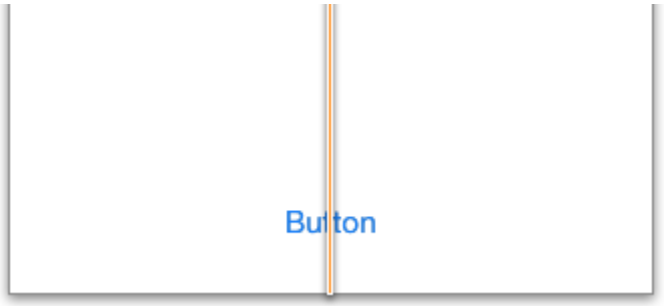
很好，T-bars 又变蓝了，布局是有效的。在 Document Outline 中，你可以看到 Horizontal Space 约束不再有一个标准的间隔了：



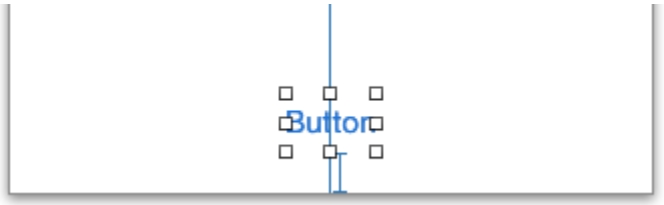
到目前为止你已经尝试过了 Horizontal Space 和 Vertical Space 约束。还有一个"center"约束。拖拽一个新的 Button 对象到画布底部中心，根据向导线完好入位：



为了保持 button 在水平方向上一直居中对齐于父视图，你需要增加一个 Center X Alignment 约束。从 Editor 菜单选择 Align\Horizontal Center in Container.这会增加一个很长的橙色线段：

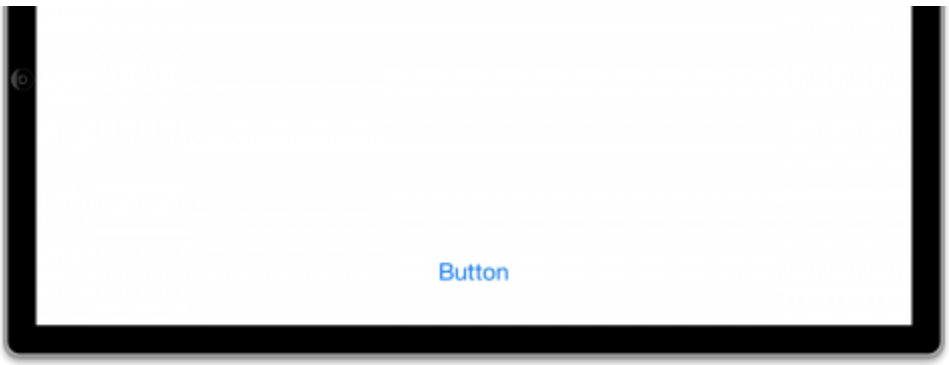


线之所以是橙色是因为你才仅仅指定了 button 的 X 轴，但 Y 轴并没有指定约束。使用 Editor\Pin 菜单在 button 和视图底部间增加一个 Vertical Space 约束。看起来像这样：



如果你不知道原因，这是 Bottom Space to Superview 选项。Vertical Space 约束使 button 远离视图底部（再一次使用标准间隔）。

运行程序并旋转到横屏模式。甚至在横屏模式，button 也保持在屏幕底部的中心：

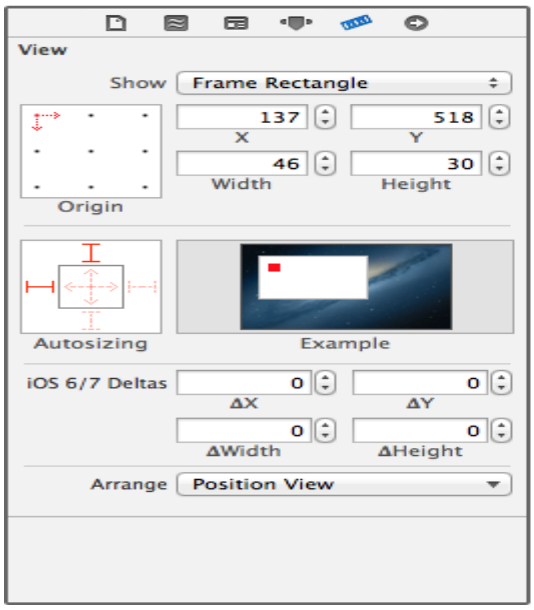


这就是你表达的意思---这个 button 始终应该位于底部中心。注意，你根本不需要告诉 Interface Builder 按钮的坐标是什么，除非你想将它固定在视图上。

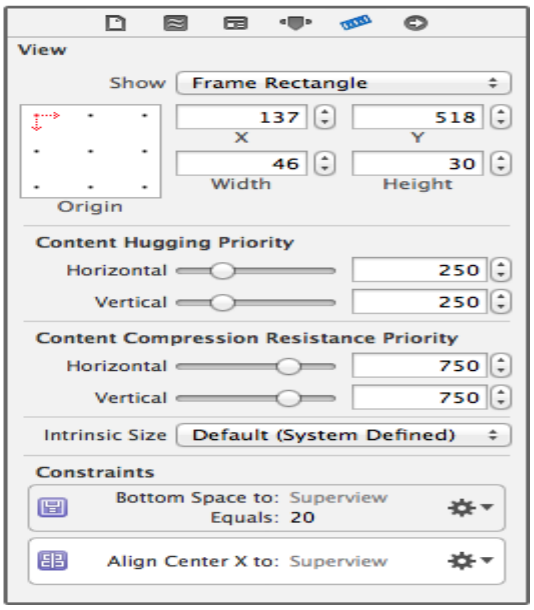
通过自动布局，你再也不需要担心视图位置的精确坐标或视图大小了。相反，自动布局会根据你设置的约束得到这两个参数。

你可以在 button 的 Size inspector 中看到这个经典转移，现在有了很大的不同：

Without Auto Layout



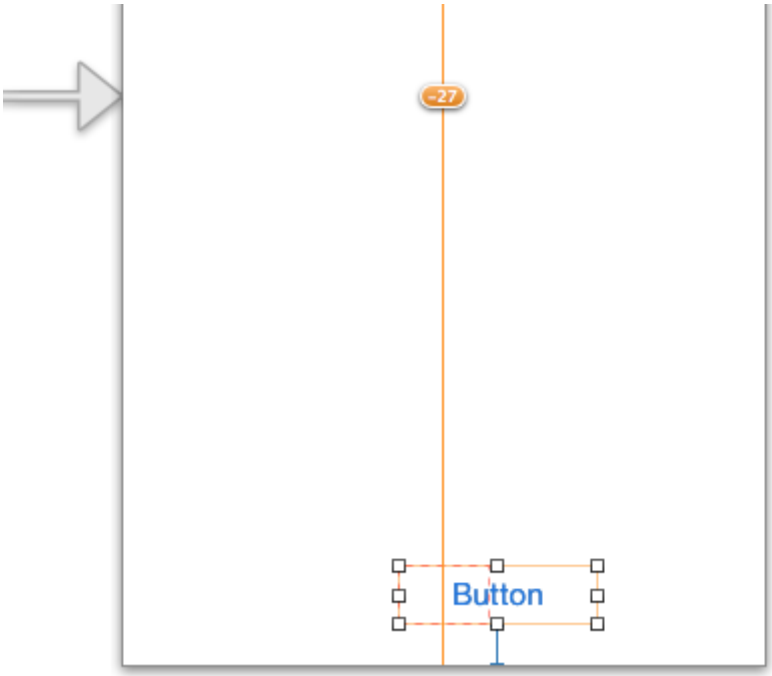
With Auto Layout



如果不使用自动布局，输入值到 X，Y，Width 或 Height 字段将会改变选中视图的位置和大小。使用自动布局后，你仍然可以输入新值到

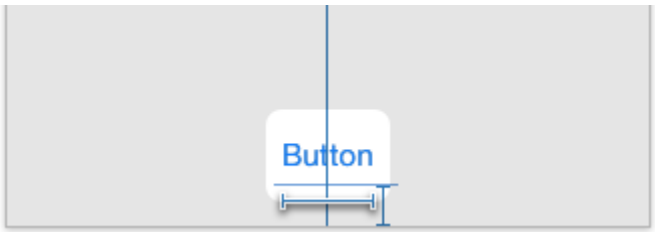
这些字段，但是如果你已经为视图设置了约束，那这可能造成视图错位。你将不得不更新约束来匹配新值。

举个例子，把 button 的宽度改为 100，画布会变成这样：

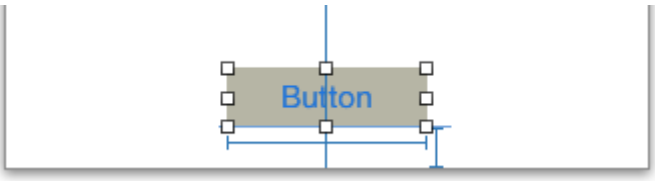


Xcode4 用 Horizontal Space 取代 Center X Alignment 约束，并且 button 上会产生一个新约束强制它的宽度为 100 points。然而，Xcode5 说，"如果你想让 button 宽度变为 100 points，对我来说无所谓，但是你要知道约束并不是这么说的。"

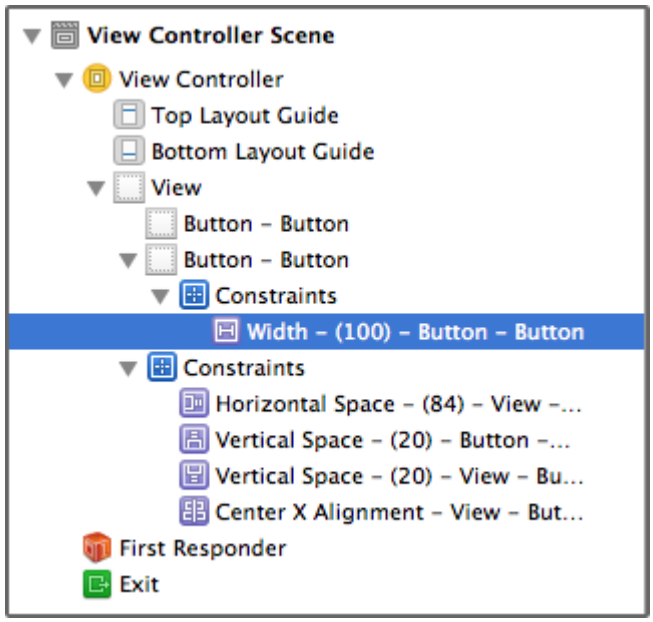
在这种情况下你希望 button 是 100 点宽。对此有一个特殊的约束类型：Fixed Width 约束。首先按一下 Undo，这样 button 又居中了，T-bars 也变蓝了。选中 button 并选择 Editor\Pin\Width。这会在 button 下面放置一个新 T-bar：



选中那个 T-bar 并在 Attributes inspector 中改变 Constant 为 100. 不管 button 的 title 多大或多小，这都会强制 button 的宽总是 100 点。为了能更好的看清你可以给 button 设置一个背景色：



你也可以在左边的 Document Outline 中看到这个新的 Width 约束：



与其他约束不同，在 button 和它的父视图之间，Width 约束只会应

用到 button 本身。你可以将这个认为是一个 button 本身和本身之间的约束。

你可能怀疑为什么 button 之前没有 Width 约束。自动布局是为何知道 button 有多宽？

事情是这样的：button 自己是知道自己有多宽。它根据自己的 title text 加上一些 padding 就行了。如果你为 button 设置一个背景图片，它也会考虑进去。

这正是我们熟悉的 intrinsic content size。并不是所有的控制器都有这个，但大部分是（UILabel 是一个例子）。如果一个视图可以计算自己理想的大小，那么你就不需要为它特别指定 Width 或 Height 约束了，你将会在稍后看到更多相关内容。

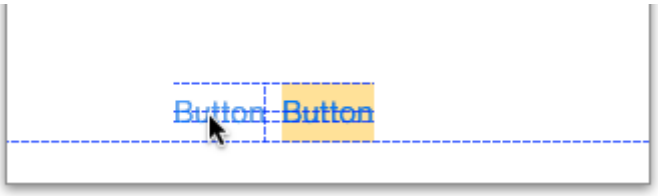
为了恢复 button 到最佳大小，首先我们需要移除 Width 约束。然后

选中 button，并从 Editor 菜单中选择 Size to Fit Content。这样就能够恢复 button 的固有的内容尺寸了。

孤掌难鸣

向导线不但出现在一个视图和它的父视图之间，而且也会出现在相同层级的视图之间。拖拽一个新的 button 到画布上进行演示。如果你将这个 button 拖近其他对象，这时他们的向导线将会开始相互影响。

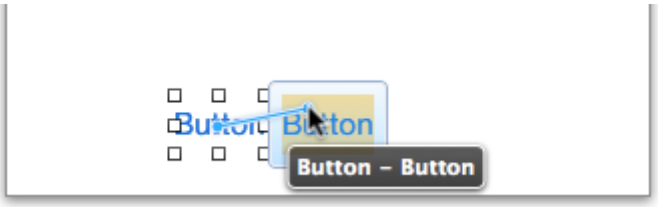
将新 button 放到之前一个 button 的后面完好入位：



这还有一些向导虚线。Interface Builder 识别出这两个 button 可以通过不同方式对齐—顶部，中心以及基线。

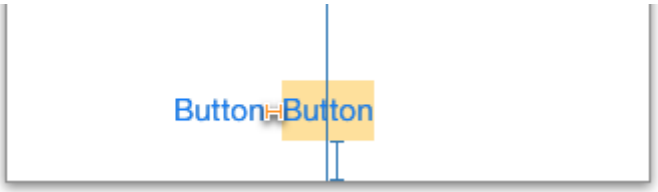
Xcode4 会将这些显著的向导线转变成新的约束。但是在 Xcode5 中，如果你想让这两个 button 间有约束，你需要自己创建。之前你已经使用过 Editor\Pin 菜单来创建这两个视图间的约束，但是还有一个更简单的方式。

选中新的 button 并按住 Ctrl 拖拽到另一个 button 上，像这样：



放开鼠标按键，出现一个弹出框。选择第一个选项，Horizontal Spacing。

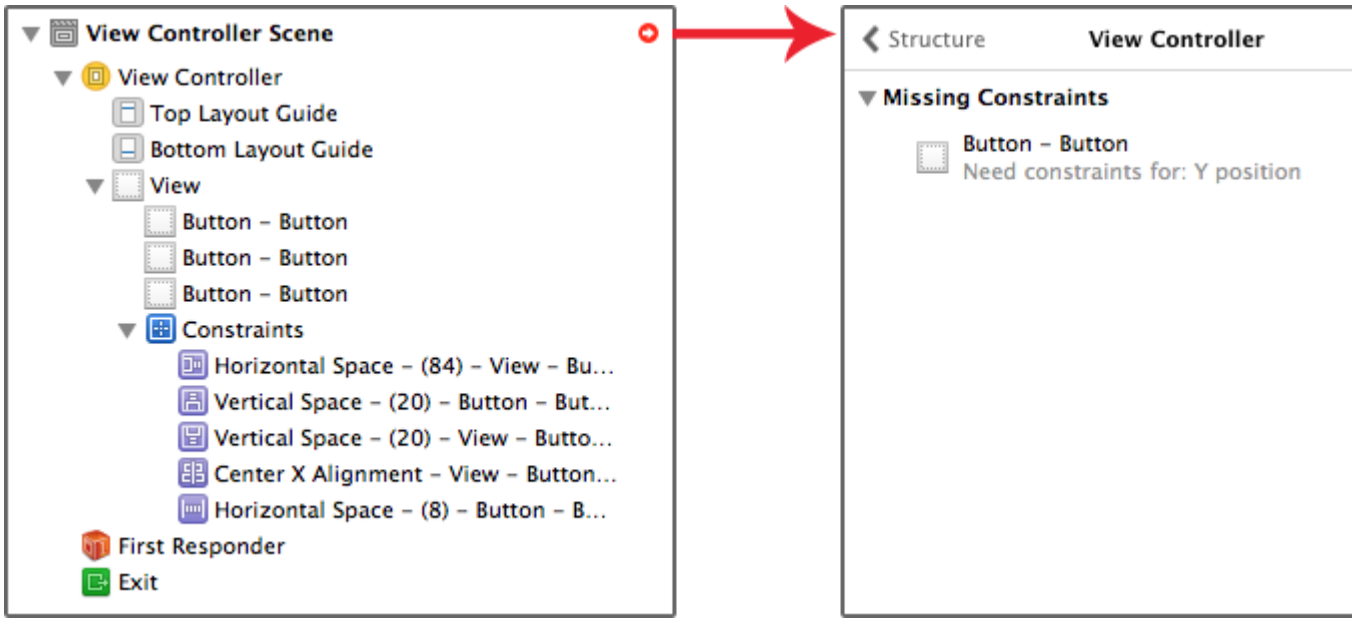
这将会创建一个新的约束：



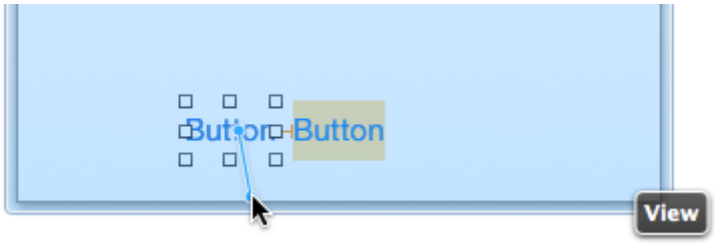
它是橙色的，这意味着这个 button 至少还需要另一个约束。button 的大小是知道的（使用 intrinsic content size），并且还有一个 button 在 X 轴上的约束。只剩下 Y 轴没有约束了。

这种缺失约束的情况是很容易确定的，但是更复杂的设计可能就没这么明显了。幸运的是，你不再需要敏思苦想，Xcode 已经记录并可以确切的告诉你缺少了什么。

在 Document Outline 中会有一个红色的小箭头，就在 View Controller Scene 后面。点击这个箭头便会看到所有 Auto Layout 问题：



我们将 Y 轴方向缺失的约束加进去。按住 Ctrl 并向下拖拽新的 button：



这次弹出菜单有不同的选项了。这次菜单的选项是基于上下文环境的——你在哪些视图间拖拽以及鼠标移动的方向。选择 Bottom Space to Bottom Layout。

现在新 button 有一个位于屏幕底部的 Vertical Space，也有一个跟其他 button 相关联的 Horizontal Space。虽然空间非常小（只有 8

points)，T-bar 可能不大容易看到，但它就在那里。

点击 Document Outline 里面的 Horizontal Space(8)：

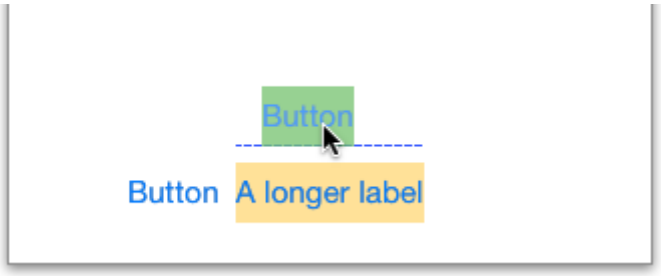
当你选中一个约束，它会高亮自己所属的控制器。这个特别的约束位于两个 button 之间。这个约束表达了：“不管第一个 button 在哪儿或多大，第二个 button 总是出现在第一个 button 的左边”。

选中黄色背景的 button 并输入较长的 label，比如：“A longer label”。输入完成后，button 会为新的 text 改变大小，并且另一个 button 会移开。

最终，它依附在第一个 button 的左边缘，这正是我们所期望的：

为了更好的摸索这是如何工作的，多练一些吧。拖拽另一个 button

到画布上并放到黄色 button 的上方，他们会垂直方向对齐到位（不要试着让两个 button 的左边缘对齐）：



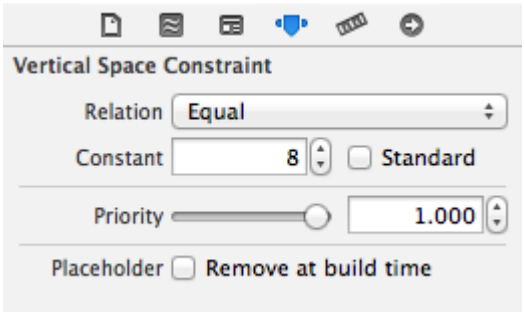
为新 button 设置一个绿色背景色，这样就可以更容易看出它的范围。

因为你将两个 button 对齐在一起，现在他们之间存在 HIG 推荐的 8 points 间隔。按住 Ctrl 在两个 button 之间拖拽将这变为一个约束。从弹出菜单中选中 Vertical Spacing。

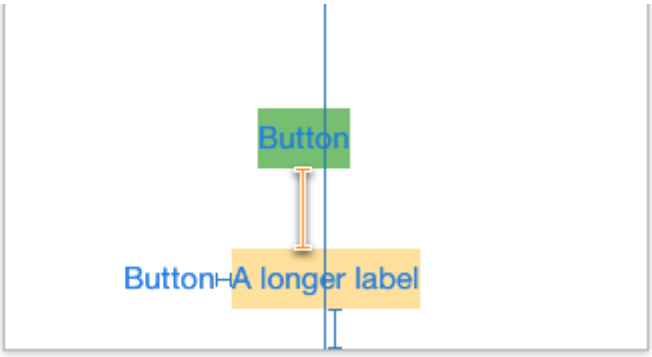
Note：“HIG”是“iOS Human Interface Guidelines”的简称，包含 Apple 推荐的良好的用户界面设计。任何 iOS 开发者都有必要读一读这个规范。HIG 解释了哪些 UI 元素适合在什么情况下使用，以及最佳使用方式。你可以在这里找到。（https://developer.apple.com/library/ios/DOCUMENTATION/UserExperience/Conceptual/MobileHIG/Introduction/Introduction.html）

然而你并没有被限制在 controls 间的标准间隔。约束是成熟的对象，就像视图一样，因此你可以改变它们的属性。

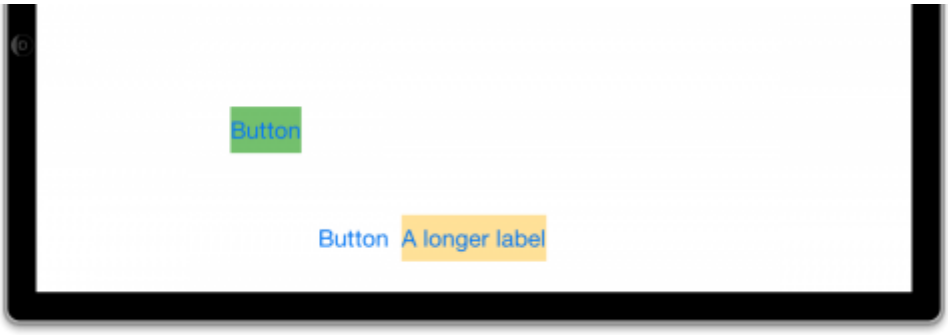
选中两个 button 之间的 Vertical Space 约束。你可以在画布上点击 T-bar，虽然这有点麻烦。目前最简单的办法就是在 Document Outline 里面选择约束。一旦你选中约束，再切换到 Attributes inspector：



在 Constant 字段里输入 40 改变约束大小。现在两个 button 更进一步的分开了，但是他们任然是连接在一起的：



运行程序并翻转到 landscape 模式查看效果：

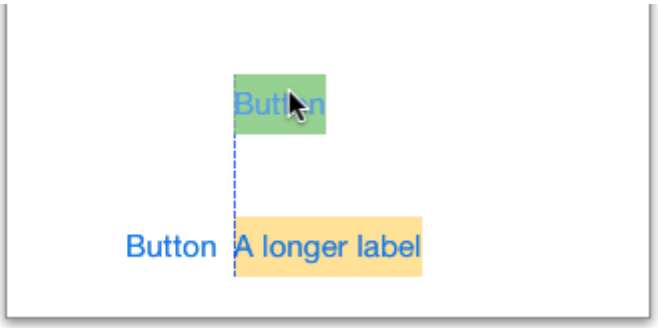


button 必然会保持他们垂直方向的排列，但是水平方向就不了！原因很明显：绿色 button 还没有 X 轴方向的约束。

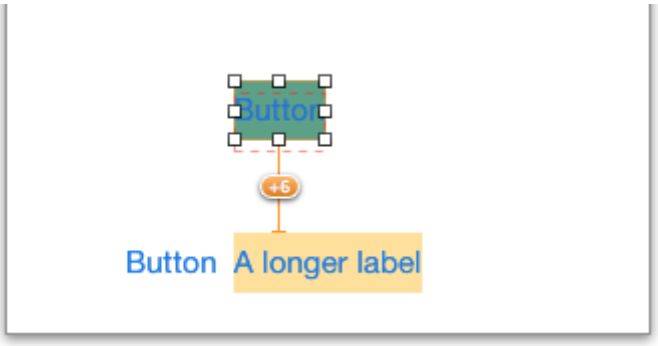
为绿色 button 增加一个到屏幕左边缘的 Horizontal Space 并不能解决问题。这样的约束只会让绿色按钮总是保持在同一个 X 轴坐标，即便是在横屏模式下。这看起来感觉不大对，所以你需要表述这样一个目的：

“黄色 button 会一直水平居中，蓝色 button 左边缘会一直跟黄色 button 左边缘对齐。”

你已经为第一种情况创建了一个约束，但是第二个并没有。Interface Builder 为对齐显示了向导线，这样你就可以将上面 button 一直拖拽到跟黄色左边缘对齐的位置：

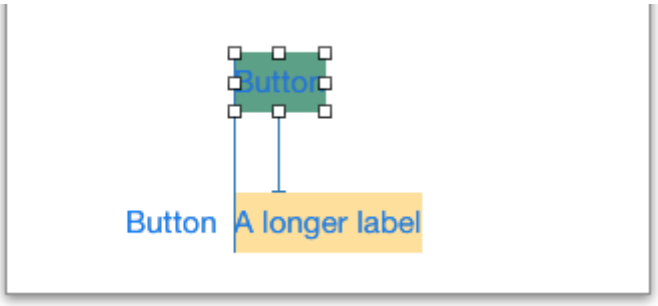


如果你也在垂直方向上拖拽 button，这时 button 框架和 Vertical Space 约束之间就不能达到正确的距离了。你在 T-bar 上将会看到橙色的 badge：

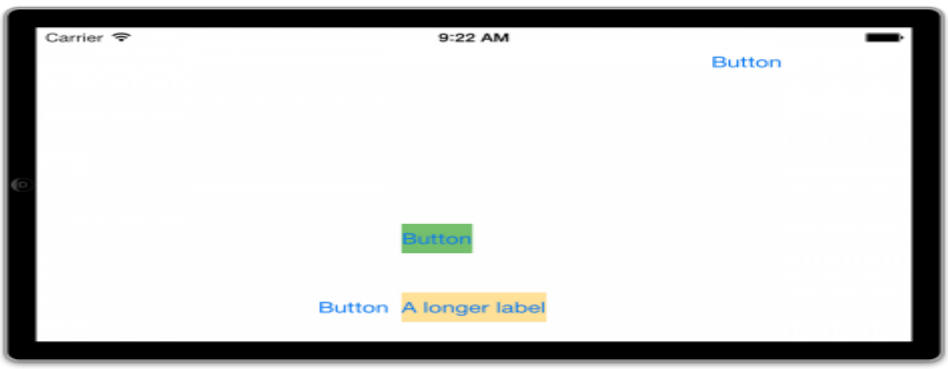


如果发生这样的情况，简单的使用方向键将 button 微调到位，直到 badge 消失。

最终，按住 Ctrl 在两个 button 间拖拽，从弹出菜单中选择 Left。这会创建一个约束：“两个视图的左边缘一直对齐”。换句话说，这两个 button 一直会有相同的 X 轴坐标。这时 T-bars 变成蓝色了：



运行程序并旋转到横屏模式：



何去何从？

现在你已经对自动布局进行了第一次尝试，感觉怎么样？这可能需要一些时间习惯，但是它能让你的工作更加简单，也会让你的 app 更加灵活。

想要学习更多的内容？继续阅读第二部分吧，你将会继续在 Interface Builder 中使用 button 进一步理解 Auto Layout 提供的多种可能性，以及你可能遇到的问题。

最重要的是，你将会使用 Auto Layout 在一个真实的程序中创建一个逼真的布局。

原文 http://www.cocoachina.com/applenews/devnews/2013/1203/7462.html?utm_source=Tuicool_Weekly

Android 实现推送方式解决方案

本文介绍在 Android 中实现推送方式的基础知识及相关解决方案。推送功能在手机开发中应用的场景是越来起来了，不说别的，就我们手机上的新闻客户端就时不 j 时的推送过来新的消息，很方便的阅读最新的新闻信息。这种推送功能是好的一面，但是也会经常看到很多推送过来的垃圾信息，这就让我们感到厌烦了，关于这个我们就不能多说什么了，毕竟很多商家要做广告。本文就是来探讨下 Android 中实现推送功能的一些解决方案，也希望能够起到抛砖引玉的作用。^_^

1. 推送方式基础知识：

在移动互联网时代以前的手机，如果有事情发生需要通知用户，则会有一个窗口弹出，将告诉用户正在发生什么事情。可能是未接电话的提示，日历的提醒，或是一封新的彩信。推送功能最早是被用于 Email 中，用来提示我们新的信息。由于时代的发展和移动互联网的热潮，推送功能更加地普及，已经不再仅仅用在推送邮件了，更多地用在我们的 APP 中了。

当我们开发需要和服务器交互的应用程序时，基本上都需要获取服务器端的数据，比如《地震应急通》就需要及时获取服务器上最新的地震信息。要获取服务器上不定时更新的信息，一般来说有两种方法：第一种是客户端使用 Pull（拉）的方式，就是隔一段时间就去服务器上获取一下信息，看是否有更新的信息出现。第二种就是 服务器使用 Push（推送）的方式，当服务器端有新信息了，则把最新的信息 Push 到客户端上。这样，客户端就能自动的接收到消息。

虽然 Pull 和 Push 两种方式都能实现获取服务器端更新信息的功能，但是明显来说 Push 方式比 Pull 方式更优越。因为 Pull 方式更费客户端的网络流量，更主要的是费电量，还需要我们的程序不停地去监测服务端的变化。

在开发 Android 和 iPhone 应用程序时，我们往往需要从服务器不定的向手机客户端即时推送各种通知消息。我们只需要在 Android 或 IPhone 的通知栏处向下一拉，就展开了 Notification Panel，可以集中一览各种各样通知消息。目前 IOS 平台上已经有了比较简单的和完美的推送通知解决方案，我会在以后详细介绍 IPhone 中的解决方案，可是 Android 平台上实现起来却相对比较麻烦。

最近利用几天的时间对 Android 的推送通知服务进行初步的研究，也希望能和大家共同探讨一下。

2. 几种常见的解决方案实现原理：

1）轮询(Pull)方式：应用程序应当阶段性的与服务器进行连接并查询是否有新的消息到达，你必须自己实现与服务器之间的通信，例如消息排队等。而且你还要考虑轮询的频率，如果太慢可能导致某些消息的延迟，如果太快，则会大量消耗网络带宽和电池。

2）SMS(Push)方式：在 Android 平台上，你可以通过拦截 SMS 消息并且解析消息内容来了解服务器的意图，并获取其显示内容进行处理。这是一个不错的想法，我就见过采用这个方案的应用程序。这个方案的好处是，可以实现完全的实时操作。但是问题是这个方案的成本相对比较高，我们需要向移动公司缴纳相应的费用。我们目前很难找到免费的短消息发送网关来实现这种方案。

3）持久连接(Push)方式：这个方案可以解决由轮询带来的性能问题，但是还是会消耗手机的电池。IOS 平台的推送服务之所以工作的很好，是因为每一台手机仅仅保持一个与服务器之间的连接，事实上 C2DM 也是这么工作的。不过刚才也讲了，这个方案存在着很多的不足之处，就是我们很难在手机上实现一个可靠的服务，目前也无法与 IOS 平台的推送功能相比。

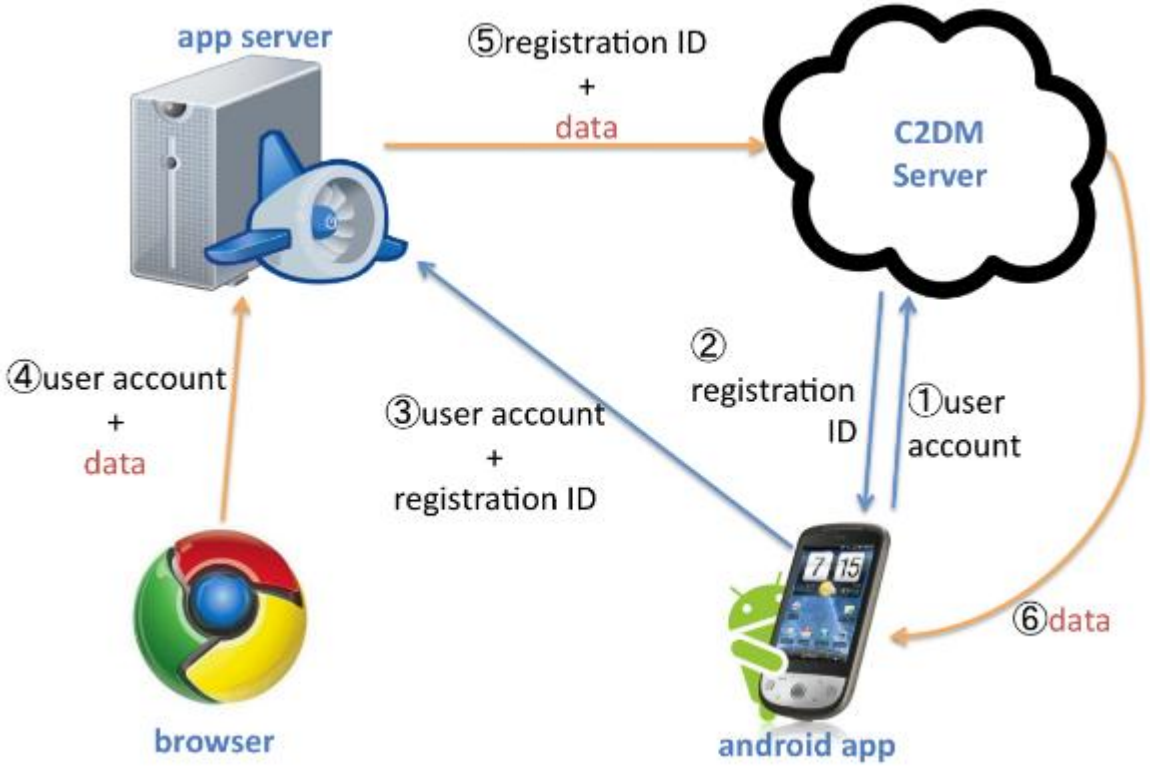
Android 操作系统允许在低内存情况下杀死系统服务，所以我们的推送通知服务很有可能就被操作系统 Kill 掉了。 轮询(Pull)方式和 SMS(Push)方式这两个方案也存在明显的不足。至于持久连接(Push)方案也有不足，不过我们可以通过良好的设计来弥补，以便于让该方案可以有效的工作。毕竟，我们要知道 GMail，GTalk 以及 GoogleVoice 都可以实现实时更新的。

3.第一种解决方案：C2DM 云端推送功能。

在 Android 手机平台上，Google 提供了 C2DM（Cloudto Device Messaging）服务，起初我就是准备采用这个服务来实现自己手机上的推送功能，并将其带入自己的项目中。

Android Cloud to Device Messaging (C2DM)是一个用来帮助开发者从服务器向 Android 应用程序发送数据的服务。该服务提供了一个简单的、轻量级的机制，允许服务器可以通知移动应用程序直接与服务器进行通信，以便于从服务器获取应用程序更新和用户数据。C2DM 服务负责处理诸如消息排队等事务并向运行于目标设备上的应用程序分发这些消息。关于 C2DM 具体使用过程，大家可以去查阅相关的资料，在这里先让我们了解下大致方案情况。

下面是 C2DM 操作过程示例图:



但是经过一番研究发现，这个服务存在很大的问题：

1）C2DM 内置于 Android 的 2.2 系统上，无法兼容老的 1.6 到 2.1 系统;

2）C2DM 需要依赖于 Google 官方提供的 C2DM 服务器，由于国内的网络环境，这个服务经常不可用，如果想要很好的使用，我们的 App Server 必须也在国外，这个恐怕不是每个开发者都能够实现的;

3） 不像在 iPhone 中，他们把硬件系统集成在一块了。所以对于我们开发者来说，如果要在我们的应用程序中使用 C2DM 的推送功能，因为对于不同的这种硬件厂商平台，比如摩托罗拉、华为、中兴做一个手机，他们可能会把 Google 的这种服务去掉，尤其像在国内就很多这种，把 Google 这种原生的服务去掉。买了一些像什么山寨机或者是华为这种国产机，可能 Google 的服务就没有了。而像在国外出的那些可能会内置。

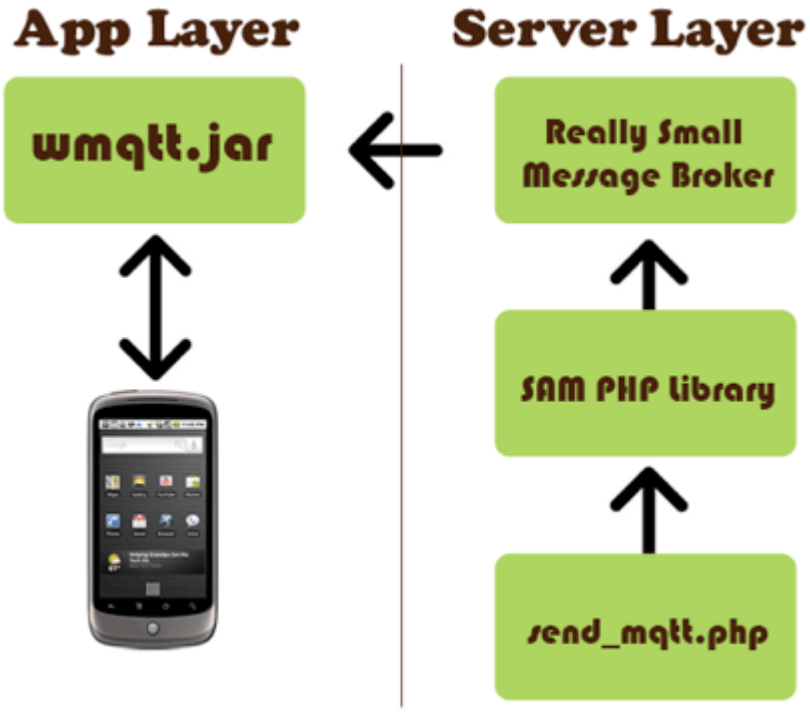
有了上述几个方面的制约，导致我最终放弃了这个方案，不过我想利用另外一篇文章来详细的介绍 C2DM 的框架以及客户端和 App Server 的相应设置方法，可以作为学习资源让我们有个参考的资料。 即然 C2DM 无法满足我们的要求，那么我们就需要自己来实现 Android 手机客户端与 App Server 之间的通信协议，保证在 App Server 想向指定的 Android 设备发送消息时，Android 设备能够及时的收到。

4. 第二种解决方案：MQTT 协议实现 Android 推送功能。

采用 MQTT 协议实现 Android 推送功能也是一种解决方案。MQTT 是一个轻量级的消息发布/订阅协议，它是实现基于手机客户端的消息推送服务器的理想解决方案。

wmqtt.jar 是 IBM 提供的 MQTT 协议的实现。我们可以从这里（https://github.com/tokudu/AndroidPushNotificationsDemo）下载该项目的实例代码，并且可以找到一个采用 PHP 书写的服务器端实现（https://github.com/tokudu/PhpMQTTClient）。

架构如下图所示：



wmqtt.jar 是 IBM 提供的 MQTT 协议的实现。我们可以从如下站点下载（http://www-01.ibm.com/support/docview.wss?rs=171&uid=swg24006006）它。我们可以将该 jar 包加入自己的 Android 应用程序中。

5.第三种解决方案：RSMB 实现推送功能。

Really Small Message Broker (RSMB) ，他是一个简单的 MQTT 代理，同样由 IBM 提供，其查看地址是：http://www.alphaworks.ibm.com/tech/rsmb。缺省打开 1883 端口，应用程序当中，它负责接收来自服务器的消息并将其转发给指定的移动设备。

SAM 是一个针对 MQTT 写的 PHP 库。我们可以从这个 http://pecl.php.net/package/sam/download/0.2.0 地址下载它.

send_mqtt.php 是一个通过 POST 接收消息并且通过 SAM 将消息发送给 RSMB 的 PHP 脚本。

6. 第四种解决方案：XMPP 协议实现 Android 推送功能。

这是我希望在项目中采用的方案，因为目前它是开源的，对于其简单的推送功能它还是能够实现的。我们可以修改其源代码来适应我们的应用程序。

事实上 Google 官方的 C2DM 服务器底层也是采用 XMPP 协议进行的封装。XMPP(可扩展通讯和表示协议)是基于可扩展标记语言（XML）的协议，它用于即时消息（IM）以及在线探测。这个协议可能最终允许因特网用户向因特网上的其他任何人发送即时消息。关于 XMPP 协议我在上篇博文中已经介绍，大家可以参考下文章：
http://www.cnblogs.com/hanyonglu/archive/2012/03/04/2378956.html

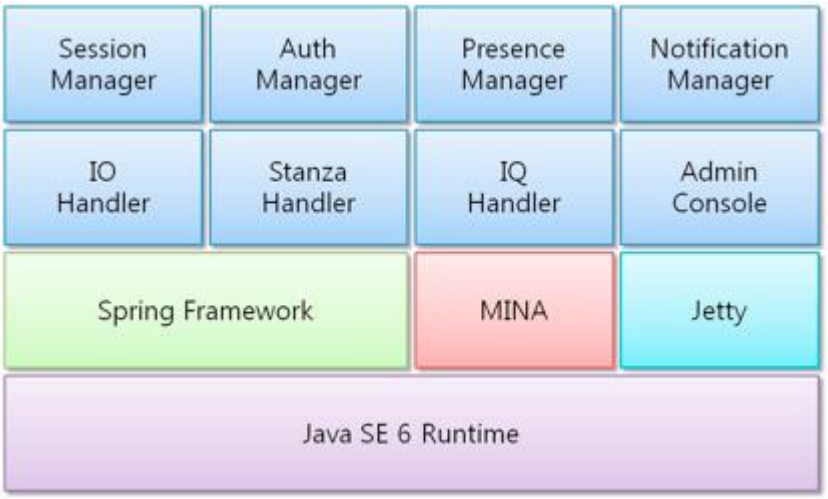
androidpn 是一个基于 XMPP 协议的 java 开源 Android push notification 实现，我会在以后的博文中详细介绍 androidpn。它包含了完整的客户端和服务器端。经过源代码研究我发现，该服务器端基本是在另外一个开源工程 openfire 基础上修改实现的，不过比较郁闷的是 androidpn 的文档是由韩语写的，所以整个研究过程基本都是读源码。

这是 androidpn 的项目主页：
http://sourceforge.net/projects/androidpn/

androidpn 实现意图如下图所示：

androidpn 客户端需要用到一个基于 java 的开源 XMPP 协议包 asmack，这个包同样也是基于 openfire 下的另外一个开源项目 smack，不过我们不需要自己编译，可以直接把 androidpn 客户端里面的 asmack.jar 拿来使用。客户端利用 asmack 中提供的 XMPPConnection 类与服 务器建立持久连接，并通过该连接进行用户注册和登录认证，同样也是通过这条连接，接收服务器发送的通知。

androidpn 服务器端也是 java 语言实现的，基于 openfire 开源工程，不过它的 Web 部分采用的是 spring 框架，这一点与 openfire 是不同的。Androidpn 服务器包含两个部分，一个是侦听在 5222 端口上的 XMPP 服务，负责与客户端的 XMPPConnection 类进行通信，作用是用户注册和身份认证，并发送推送通知消息。另外一部分是 Web 服务器，采用一个轻量级的 HTTP 服务器， 负责接收用户的 Web 请求。服务器架构如下：



最上层包含四个组成部分，分别是 SessionManager，Auth Manager，PresenceManager 以及 Notification Manager。SessionManager 负责管理客户端与服务器之间的会话，Auth Manager 负责客户端用户认证管理，Presence Manager 负责管理客户端用户的登录状态，NotificationManager 负责实现服务器向客户端推送消息功能。

这个解决方案的最大优势就是简单，我们不需要象 C2DM 那样依赖操作系统版本，也不会担心某一天 Google 服务器不可用。利用 XMPP 协议我们还可以进一步的对协议进行扩展，实现更为完善的功能。 采用这个方案，我们目前只能发送文字消息，不过对于推送来说一般足够了，因为我们不能指望通过推送得到所有的数据，一般情况下，利用推送只是告诉手机端服务器发生了某些改变，当客户端收到通知以后，应该主动到服务器获取最新的数据，这样才是推送服务的完整实现。 XMPP 协议书相对来说还是比较简单的，值得我们进一步研究。

但是在经过一段时间的测试，我发现关于 androidpn 也存在一些不足之处：

1. 比如时间过长时，就再也收不到推送的信息了。
2. 性能上也不够稳定。
3. 如果将消息从服务器上推送出去，就不再管理了，不管消息是否成功到达客户端手机上。

等等，总之，androidpn 也有很多的缺点。如果我们要使用 androidpn，则还需要做大量的工作。

至于详细使用过程，我们会在下个博文中再给大家介绍。

7.第五种解决方案：使用第三方平台。

第三方平台有商用的也有免费的，我们可以根据实现情况使用。关于国内的第三方平台，我感觉目前比较不错的就是极光推送。关于极光推送目前是免费的，我们可以直接使用。关于详细情况，大家可以查看它的主页：http://www.jpush.cn/index.jsp，这里不再详细描述。

关于国外的第三方平台我也见过几个：http://www.push-notification.org/。有兴趣的朋友可以查阅相关信息。使用第三方平台就需要使用别人的服务器，关于这点，你懂的。

8.第六种解决方案：自己搭建一个推送平台。

这不是一件轻松的工作，当然可以根据各自的需要采取合适的方案。

好了，以上是关于在 Android 中实现推送方式的基础知识及相关解决方案。

参考：

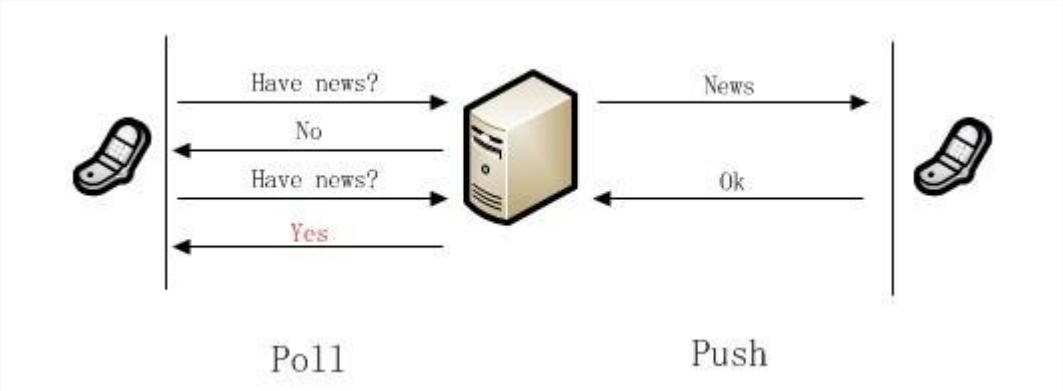
http://www.infoq.com/cn/articles/baidu-android-cloud-push

http://www.cnblogs.com/hanyonglu/archive/2012/03/04/2378971.html

一、推送服务简介

消息推送，顾名思义，是由一方主动发起，而另一方与发起方以某一种方式建立连接并接收消息。在 Android 开发中，这里的发起方我们把它叫做推送服务器（Push Server），接收方叫做客户端（Client）。相比通过轮询来获取新消息或通知，推送无论是在对客户端的资源消耗还是设备耗电量来说都比轮询要好，

所以，目前绝大多数需要及时消息推送的 App 都采用 Push 的方式来进行消息通知。



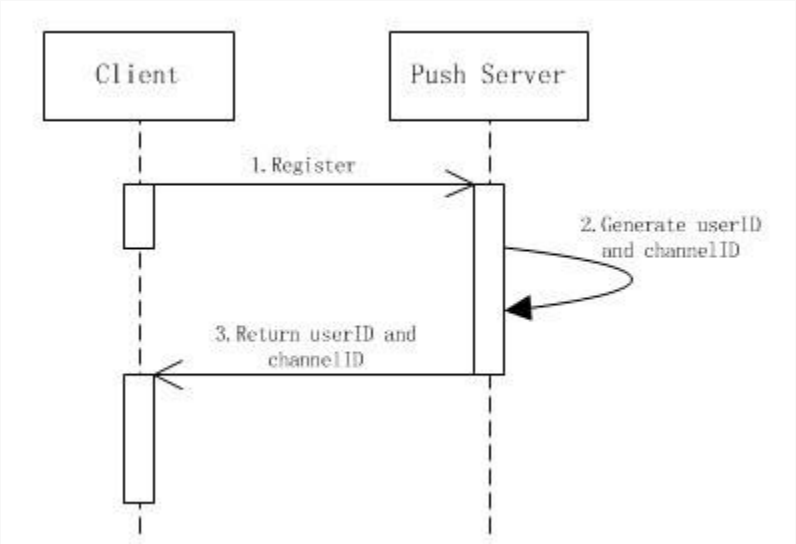
身在天朝，置身墙内！Android 生态系统原本提供了类似于 Apple iOS 推送服务 APNS 的 GCM(Google Cloud Messaging for Android)，以前叫 C2DM,但是由于某些原因，导致这项服务在国内不是很好使，为了弥补这个不足，并且我朝各大同胞又想使用 Android 推送服务，所以国内各大平台陆续推出了 GCM 的替代品，今天要介绍的就是其中一家，由百度提供的云推送。另外，国内做消息推送服务的还有极光推送和个推等，他们的客户包括新浪微博、淘宝等国内一线大公司。推送的实现技术简单来说就是利用 Socket 维持 Client 和 Server 间的一个 TCP 长连接，通过这种方式能大大降低由轮询方式带来的 Device 的耗电量和数据访问流量。目前，百度云推送提供的推送服务支持的单一消息体大小是 4k，如果超过 4k，则建议在消息内携带服务请求 URL 进行二次请求。目前，百度云推送针对 Android 端提供通知推送，文本消息推送以及富媒体推送。

二、使用场景

1. 单播消息推送

Push Server 向指定的设备（Device）或是用户（User）推送消息，一个用户对应一个 userID，一个 User 可能拥有多台 Device，我们希望向同一个 userID 推送消息时，他所有绑定了 userID 的 Device 都能收到消息。百度云推送给出的解决方案是通过 Client 向 Push Server 注册，并在 Client 端的监听端口取得 Push Server 返回的 channelID 和 userID，channelID 指定一个终端，在向 Push Server 注册的过程中,Device 可以发送 IMIE 码或者 UUID 作为唯一标示,在 Push

Server 注册后再返回给 Client 生成的 `channelID` 和 `userID`。这两个 ID 获取到后由开发者自行维护，注册完毕后，Push Server 维护一个注册设备列表，这个列表维护了 `userID` 和 `channelID` 以及与 Device 对应的关系，当需要向指定的设备或用户推送消息时，Push Server 会首先遍历这个设备列表，通过这两个 ID 来做唯一性判断并找到需要推送消息的 Device，然后就可以进行消息推送了。



实例：用户 A 发表问题时，记录问题 id 及其对应的 A 的 userID（或 channelID），用户 B 发表问题回答时，通过服务端 API 向问题 id 对应的 userID（或 channelID）指向的 Device 推送答案。

2. 分组消息推送

百度云推送通过对 Client 设置标签（Tag）的方式来进行用户分组，Tag 的产生方式可以是由 Client 维护也可以由 Server 收集，Push Server 针对不同的 Tag 进行推送过滤，最终将消息推送到指定的 Client。无论是由 Client 主动设置的 Tag 还是由 Server 根据用户使用习惯收集的，都由 Push Server 进行统一管理，在基于 Tag 的分组消息推送实现上，Push Server 首先根据指定 Tag 从所有 Tag 下遍历出的对应的已注册的 Device，从而可以获得与 Device 对应的 `userID` 和 `channelID`，继而可以针对指定 Tag 进行分组消息推送。对比单播消息推送，分组消息推送在推送周期上势必要长一些，并且在待推消息列表的维护上也需要做一些处理，哪些消息是推送成功的，哪些是失败的，这需要接收消息推送的

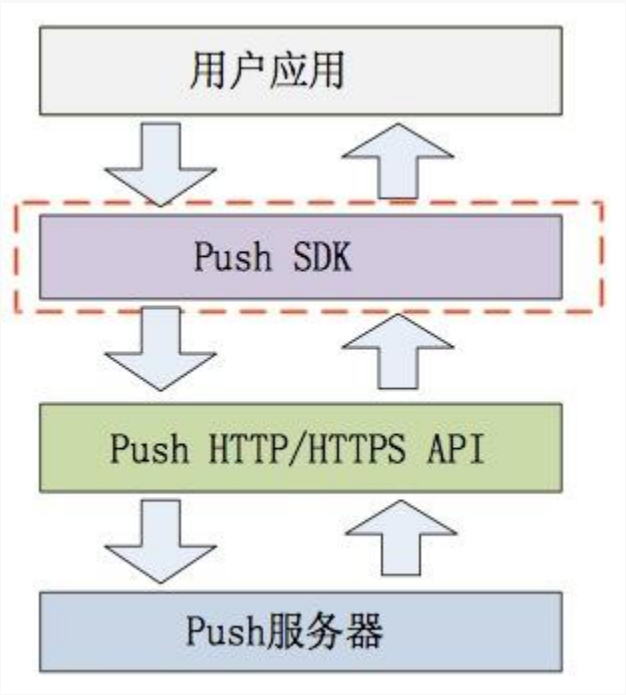
Client 在接收到消息后给 Push Server 一个消息回执，这样就保证了消息送达的准确性，如果消息推送失败，则分组列表里的待推消息会继续推送，直到推送消息成功。另外，在消息推送的实时性上，分组消息推送对比单播消息推送会根据分组消息队列的先后存在一个消息接收的延时，好比现在微信公众账号的推送，就是一个分组消息推送的实例，在消息接收的时效性上对比单播推送存在一定的延时性。
另外，还有一类消息推送使用场景，就是广播消息，该类型可以理解为分组消息的一个特列，即向所有的 Tag 对应的 Client 推送消息。广播消息是对全体集合的一个消息推送，在消息队列维护和消息推送时效性上比单个或几个 Tag 的分组推送成本要高。

实例：给应用提供喜好设置页面，用户勾选不同的类别，触发对应 Tag 的设置，这种方式是由 Client 主动维护 Tag。或者用户阅读了某个类别的图书，触发对应 Tag 的设置，在服务端，给指定类别的图书设置 Tag，后续会根据服务端收集的 Tag 给应用推送该 Tag 下的新书信息，这种方式就是由服务端来维护 Tag 分组。

三、百度云推送 Android_SDK

百度提供了完整的 Demo 帮助开发者集成云推送服务，推送服务 SDK 通过.jar 包和.so 文件的方式可以集成到我们自己的工程中。在此之前，需要到百度开发者中心进行应用注册并获取 API Key，这个作为使用推送服务应用的唯一标示，具体流程我就不赘述了，需要使用的话可以直接访问百度开发者中心进行查看。

下面主要看看 Android_SDK 的整体概览和内部运行机制：



上图是百度云推送 Android_SDK 的框架图，通过 SDK 可以绕过复杂的 Push HTTP/HTTPS API 直接和 Push 服务器进行交互，主要提供如下功能：

- Push 服务初始化以及 Client 注册绑定
- 创建或删除标签（Tag）
- 接收 Push Server 的通知并提供自定义展现消息方式
- 推送统计分析功能，包括通知的点击和删除统计以及应用使用情况统计
- 富媒体推送

在 Android 端，总共实现了三个 Receiver 和一个 Service，其中，一个 Receiver 是用来处理注册绑定后接收服务端返回的 channelID 等信息：

```
<receiver
android:name="com.baidu.android.pushservice.RegistrationReceiver"
android:process=": bdservice_v1">
<intent-filter>
<action android:name="com.baidu.android.pushservice.action.METHOD "
/>
<action
android:name="com.baidu.android.pushservice.action.BIND_SYNC " />
</intent-filter>
<intent-filter>
<action android:name="android.intent.action.PACKAGE_REMOVED"/> <data
```

```
android:scheme="package" />
</intent-filter>
</receiver>
```

第二个 Receiver 是用于接收系统消息以保证 PushService 正常运行：

```
<receiver
android:name="com.baidu.android.pushservice.PushServiceReceiver"
android:process=": bdservice_v1">
<intent-filter>
<action android:name="android.intent.action.BOOT_COMPLETED" />
<action android:name="android.net.conn.CONNECTIVITY_CHANGE" />
<action
android:name="com.baidu.android.pushservice.action.notification.SHOW
" /> <action
android:name="com.baidu.android.pushservice.action.media.CLICK" />
</intent-filter>
</receiver>
```

第三个 Receiver 就是开发者自己实现的用来接收并处理推送消息：

```
<receiver android:name="your.package.PushMessageReceiver">
<intent-filter>
<!-- 接收 push 消息 -->
<action android:name="com.baidu.android.pushservice.action.MESSAGE"
/>
<!-- 接收 bind、setTags 等 method 的返回结果 -->
<action android:name="com.baidu.android.pushservice.action.RECEIVE"
/>
</intent-filter>
</receiver>
```

一个 Service 就是在后台运行的用于保障与 Push Server 维持长连接并做相关处理的后台服务：

```
<service android:name="com.baidu.android.pushservice.PushService"
android:exported="true" android:process=" bdservice_v1"/> <!-- push
service end -->
```

在开发者自己需要处理的广播接收器中，可以对接收到的推送消息进行处理，Push 消息通过 action 为 com.baidu.android.pushservice.action.MESSAGE 的 Intent 把数据发送给客户端 your.package.PushMessageReceiver，消息格式由

应用自己决定，PushService 只负责把服务器下发的消息以字符串格式透传给客户端。接口调用回调通过 action 为 com.baidu.android.pushservice.action.RECEIVE 的 Intent 返回给 your.package.PushMessageReceiver。

PushMessageReceiver.java

```
/**
 * Push 消息处理 receiver
 * @Author Ryan
 * @Create 2013-8-6 下午 5:59:38
 */
public class PushMessageReceiver extends BroadcastReceiver {
  public static final String TAG =
PushMessageReceiver.class.getSimpleName();

  @Override
  public void onReceive(final Context context, Intent intent) {

      if (intent.getAction().equals(PushConstants.ACTION_MESSAGE)) {
          //获取消息内容
          String message = intent.getExtras().getString(
                  PushConstants.EXTRA_PUSH_MESSAGE_STRING);
          //消息的用户自定义内容读取方式
          Log.i(TAG, "onMessage: " + message);

      } else if
(intent.getAction().equals(PushConstants.ACTION_RECEIVE)) {
          //处理绑定等方法的返回数据
          //PushManager.startWork()的返回值通过
PushConstants.METHOD_BIND 得到

          //获取方法
          final String method = intent
                  .getStringExtra(PushConstants.EXTRA_METHOD);
          //方法返回错误码。若绑定返回错误（非 0），则应用将不能正常
接收消息。
          //绑定失败的原因有多种，如网络原因，或 access token 过期。
          //请不要在出错时进行简单的 startWork 调用，这有可能导致死循
环。
```

```
            //可以通过限制重试次数，或者在其他时机重新调用来解决。
            final int errorCode = intent
                    .getIntExtra(PushConstants.EXTRA_ERROR_CODE,
                            PushConstants.ERROR_SUCCESS);
            //返回内容
            final String content = new String(
                    intent.getByteArrayExtra(PushConstants.EXTRA_CONTENT));

            //用户在此自定义处理消息,以下代码为 demo 界面展示用
            Log.d(TAG, "onMessage: method : " + method);
            Log.d(TAG, "onMessage: result : " + errorCode);
            Log.d(TAG, "onMessage: content : " + content);

        }
    }

}
```

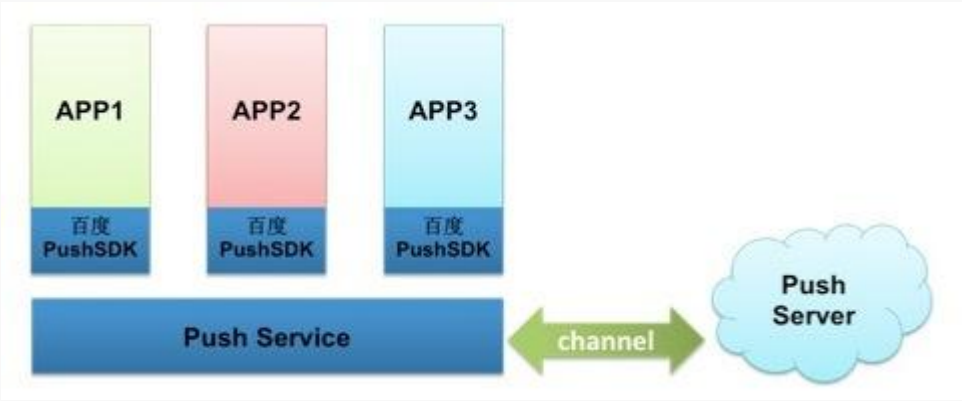
通过在入口 Activity 的 onCreate 方法中进行推送服务的注册绑定后，即可在推送管理后台或是自己的应用服务器上进行消息推送的操作了。

```
PushManager.startWork(getApplicationContext(),PushConstants.LOGIN_TYPE_API_KEY, "you_api_key");
```

另外，云推送提供 php、java 等 Server 端的 SDK 供开发者在自己的服务器上实现推送服务进行定制化管理和操作。

四、单服务单通道机制

百度云推送实现了单服务单通道的机制，如果在一台 Device 上安装了多款 Push SDK 的应用，不会为每个应用都创建 PushService，而是会采用多应用共享一个 PushService 的模式。这样既能减少资源消耗也能降低网络流量。PushService 运行于一个独立进程，没有和主进程运行于同一进程，所以主进程不需要常驻内存，当有新的 Push 消息时，PushService 会通过 Intent 发送消息给主进程进行处理。通过 Intent，以指定目标应用包名的方式，发送私有消息给应用。应用即不能接收不属于自己的消息，也不能截取别人的消息，同时又降低了消耗，如下为示意图：



后记：如今，国内提供 Android 推送服务的还有很多家，例如个推和极光推送等，实现的原理大同小异，开发者可以根据自身需要进行选择。身在天朝，置身墙内，用不到 GCM，就创造 Android Push Service for China 自给，或者，出走！

原文
http://blog.csdn.net/webgeek/article/details/17165751?utm_source=Tuicool_Weekly

James Reading

有一个多月没有整理阅读的内容了，最近一段时间在阅读几本书，《Systems-Performance-Enterprise-and-the-Cloud》 By Gregg Brendan,《The Art of Computer System Performance》,《NoSQL Distilled》 都很不错， 不过都没看完， 就不在这里多说了。 哈哈。

下面是 10-14 到今天 12-07 为止阅读过， 并认为值得了解的内容。

运维与监控

http://t.cn/zR5ADrP 预警是关于“Unknown unknown”，在文中，Baron Schwartz 介绍了监控系统的几个基本原则：1. 从业务角度去监控，每秒钟处理多少业务量，处理的速度如何，2. 度量并分析你关心的指标，3. 永远不要针对无法修复的问题做告警，比如 MySQL 的备库延迟，比如因为备份而引起的负载增加。

http://t.cn/zRtymSO 构建有效的告警系统，1. 针对哪些指标进行监控（超载监控，频次监控，时间窗口统计监控），2.如何定义正常状态(Normal behavior)，3.如何定义非正常状态（Abnormal or anomalies ）。

http://t.cn/zRtUpuy 性能与可运维性模式，这篇文章是这本书的深度书评，从这篇书评看，本书几乎涉及到我接触/了解到的可运维性的大部分内容，对于性能相关内容，介绍比较粗浅。

http://t.cn/zR6FCfp “Monitor Some of the Things”， http://t.cn/zR6FCfN “What should I Monitor”,Baron Schwartz 最近做的两个关于如何监控/监控什么/如何 Alert/如何发现 Anomaly/如何做容量规划/如何做基本的性能诊断.

http://t.cn/zR6HmCy Chaos Kong, Netflix 的地区级容灾工具, 文章要点: 1. Chaos Monkey 负责单主机故障容灾,Chaos Gorilla 负责 Availability Zone 的容灾,Chaos Kong 负责地区级故障容灾, 2. 利用 Amazon 的预留主机策略来实施地区级容灾,3. 自己实现 CDN(21 个机房),4.开发负责到底,5. 一切都保留 3 份冗余

http://t.cn/zRjiPwZ netflix 的自动容量控制平台-scryer, 基于历史的负载特征, 做容量与资源的拟合, 再基于此拟合自动的通过 AWS 的工具上下线机器, 从而更加有效的使用机器资源, 降低成本. 当然, 并不是所有负载都是基于固定模式(pattern)的, 此系统也接受人工指定负载特征,来应对节假日模式.

http://t.cn/zRtU6xQ Netflix 的 Hystrix 高容错系统的介绍。重点介绍了他们的监控系统，以及参造《Release IT！》一书中介绍的几种提高系统可靠性的 Stability Patterns，如 Bulkheads，Circuit Breaker，Threadpool 与 Semophore 来控制并发访问，使用 Failback、Fast Fail 模式来控制故障蔓延。

http://t.cn/zR6s4f3 “How to Run a Post-Mortem With Humans”, 如何实现 no-shame 的故障事后分析, 1. 从心理学的角度分析人的认知, 人通常都会因故障从自身触发而感到 Shame, 2. 默认情况下, 我们都倾向于将故障定位为人的问题, 是人不够仔细不够小心, 而这对于后续如何避免故障作用不大, 3. 更好的方式是, 假设人会犯此类错误并从机制上避免

性能优化

http://t.cn/zRyirMZ iconfinder 如何优化他们的页面处理引擎(Render),将页面的处理时间从开始的 91ms->29ms->20ms, 而丝毫不涉及到传统上大家认为的,系统慢是源自数据库慢. 哈哈. 基本的观点是: 需要通过 Profiling 的方法找到系统慢的地方,并作针对性的优化,而不只是找个替罪羊.

http://t.cn/zHuwByv Brendan Gregg 的主页, 他的几乎所有的演讲 ppt, 大部分比较好的文章,在此都有汇集; 另, 看了下他推荐的阅读列表, 大部分是关于性能分析,性能优化, 容量规划与分析的书籍, 值得参考下.

http://t.cn/zR6Q4RF Brendan Gregg 的新书 Systems-Performance-Enterprise-and-the-Cloud 已经上架,可以购买了(价格较高,谨慎动手), 主要内容: 1. 优化的方法论(术语/概念/模型/方法与技术),

2. 动态追踪技术与工具，包含 Dtrace/Systemtap/Perf，3. 系统各个组件的优化技术，4. 压测,如何避免常见误区

http://t.cn/8khwx3h http://t.cn/8khwx3P 如何利用 HyperLogLog 算法在 Oracle 中(增量的)计算表上的 distinct 值，HyperLogLog 是一种基于 Hash 桶计算近似 Distinct 值的算法，计算的精度基本在+-2%的范围，优势有三：1. 内存耗费非常小，2. 计算速度快，3. 可以增量计算. 这两篇文章是介绍性的，不过很清晰

http://t.cn/SxMn6K 深入探讨 Java 语言的各种对象(Collection)/类型在运行时的内存消耗，各种 Collection 对象的内存消耗对比，如何更好的管理对象的生命周期(Life Cycle)，如何有效的利用 Java 的 Heap 空间，同时又不降低处理的性能.

http://t.cn/8kLSAdm 不同 Redo Size 在 Exadata SSD 上的效果，Redo Size 越大，log write 的写入时延波动性越大，从而越不可用. 【从我个人的经验看，1. 尽可能不要使用 SSD 作为 Redo 写，2.如果使用，a. 专用设备,b. 足够的预留空间(高 OP)，3. 使用成熟厂家的产品(如 Fusion-IO,Intel)，GC 算法至关重要】

http://t.cn/8ktRmee 不错的关于 SSD 以及 IO 相关的小提示. RethinkDB 出品.

http://t.cn/8kfhwWq 针对 EMC XtremIO 的文章 http://t.cn/8kcP1i5 的回击. 从我的角度理解，EMC 确实没有说清楚他的优势，或者说，在 GC 这件事情上，他没有做什么事情，而 GC 对于每一个 Flash 厂商来讲都应该是重要的事情.

架构,评论与其它

http://t.cn/zRtV8xe Innotop 工具的介绍，简单的登录配置，如何配置 MySQL 集群,通过 innotop 检测一个集群的状态，比如一个 master 多个 Slave 的情况，如何通过 Innotop 来管理集群的多台机器,在权限允许的情况下,可以通过它对多台机器发出命令. 功能还是很强大的.

http://t.cn/zR6QhLX Todd Hoff 对 sosp 2013 论文的回顾，“关于同步,所有你需要知道的内容”，要点：1. 锁的利用需要基于硬件平台，以及对应的工作负载特征进行选择,2. 同步操作的可扩展性主要是硬件的一个属性，3. 同步操作在单 CPU 上扩展性最好,4.9 种不同的锁算法各有其合适的场景，也即合适才是最好的.

http://highscalability.com/blog/2013/12/4/how-can-batching-requests-actually-reduce-latency.html 批处理为何可以,以及在什么样的情况下,可以降低时延.

http://t.cn/8kLY6dz 为什么 Oracle 不会杀掉 MySQL,1. MySQL 并不是 Oracle 的直接竞争产品,Oracle 的客户主要为运行企业级软件的企业客户,而不是互联网客户. 2. Oracle 是为了硬件而购买 Sun,也即目标是 Exa 系列的产品,3. Larry

的目标是钱,并不反对开源,4.MySQL 的企业支持业务发展也不错,5. M 有替代的竞争产品

http://t.cn/8kA9brd 硬盘的生命周期到底是怎么样的？一块普通的磁盘寿命如何?

http://t.cn/zjjwPQ1 为什么 MongoDB 在 Etsy 使用的并不好，MongoDB 的成名/成功主要得益于两大功能：Schema-Free,Auto-Sharding, 当一个公司已经有相对成熟的 MySQL 运维体系的时候, 当公司有足够的技术能力做好去范式化/Sharding Key 的选择/自动 Sharding 扩展这些功能时, 引入一个新的数据库好处就很有限了

http://t.cn/zRQV5qv Robin Harris (StorageMojo)谈论传统大型存储的情况,1. 已经不适合时代, 2. 会逐渐被 Flash Storage 取代. 仍然健在的原因,1.【Availability】更好的可用性，2. 更友好的使用。Flash 存储的工作方向：1. 更好的压缩，2. 更好的实时去重，有效提供更好的 Capacity/$。

http://t.cn/zRIw0pV (请自备梯子), Twillo 的高可用架构变迁, 谈及 Twillo 如何根据业务的需求与特征, 并从故障的角度分析高可用的天敌: 数据持久化与变更管控, 总结了几条规则: 1. 尽可能无状态,2.分离有状态与无状态的组件,3.使用 Cache 与 Sharding,4.分解数据的生命周期,降低数据管理复杂度.

http://t.cn/zRJMybK MySQL 上的几种高可用方案, 各种对比, 各种介绍, 自己体会吧.

社会心理

人力资本与非人力资本在产权性质上的差别很大,在自由社会中,人力资本的所有权仅限于他本人. — 罗森(芝加哥大学经济系,劳动力经济学的领导人物).

果树会结果,农地有收成,结果与收成都是收入. 然而, 这收成可不是在果熟或稻熟时才得到的. 果树或农作物每天都在变, 不停地变, 而每一小变都是收入(或负收入), 所以, 收入是一连串的事件了. －摘自周其仁《收入是一连串事件》之张五常序.

The first principle is that you must not fool yourself - and you are the easiest person to fool — By Richard Feynman.

http://t.cn/zR6zfjL 控制人们的头脑是控制整个国家的关键，语言文字就是制度的基石。 ——〔波兰〕切斯瓦夫·米沃什

http://t.cn/8kLAGW1 权利是得到社会认可的、大部分人主动维护的选择的自由。任何在现实中能够行使的权利，都离不开他人的背书和支持。换言之，我们可以倡议某种权利，并声称它是一种 “自然权利” 或 “天赋权利”，但除非它得到普遍的尊重和维护，它就只是应然而非实然的关于权利的主张而已

原文 http://www.dbthink.com/archives/908?utm_source=Tuicool_Weekly

深入剖析阿里巴巴云梯 YARN 集群

《程序员》杂志 2013 年 11 月刊 HadoopYARNMapReduceHDFS 阿里巴巴云梯集群 Spark

摘要:阿里巴巴是国内使用 Hadoop 最早的公司之一,已开启了 Apache Hadoop 2.0 时代。本文将详细介绍阿里巴巴如何充分利用 YARN 的新特性来构建和完善其多功能分布式集群——云梯 YARN 集群。

阿里巴巴作为国内使用 Hadoop 最早的公司之一，已开启了 Apache Hadoop 2.0 时代。阿里巴巴的 Hadoop 集群，即云梯集群，分为存储与计算两个模块，计算模块既有 MRv1，也有 YARN 集群，它们共享一个存储 HDFS 集 群。云梯 YARN 集群上既支持 MapReduce，也支持 Spark、MPI、RHive、RHadoop 等计算模型。本文将详细介绍云梯 YARN 集群的 技术实现与发展状况。

MRv1 与 YARN 集群共享 HDFS 存储的技术实现

以服务化为起点，云梯集群已将 Hadoop 分为存储（HDFS）服务与计算（MRv1 和 YARN）服务。两个计算集群共享着这个 HDFS 存储集群，这是怎么做到的呢？

在引入 YARN 之前，云梯的 Hadoop 是一个基于 Apache Hadoop 0.19.1-dc 版本，并增加许多新功能的版本。另外还兼容了 Apache Hadoop 0.19、0.20、CDH3 版本的客户端。为了保持对客户端友好，云梯服务端升级总会保持对原有客户端的兼容性。另外，为了访问数

据的便捷性，阿里的存 储集群是一个单一的大集群，引入 YARN 不应迫使 HDFS 集群拆分，但 YARN 是基于社区 0.23 系列版本，它无法直接访问云梯 HDFS 集群。因此实现 YARN 集群访问云梯的 HDFS 集群是引入 YARN 后第一个需要解决的技术问题。

Hadoop 代码主要分为 Common、HDFS、Mapred 三个包。

- Common 部分包括公共类，如 I/O、通信等类。
- HDFS 部分包括 HDFS 相关类，依赖 Common 包。
- Mapred 部分包括 MapReduce 相关代码，依赖 Common 包和 HDFS 包。

为了尽量减少对云梯 HDFS 的修改，开发人员主要做了以下工作。

- 使用云梯的 HDFS 客户端代码替换 0.23 中 HDFS，形成新的 HDFS 包。
- 对 0.23 新的 HDFS 包做了少量的修改使其可以运行在 0.23 的 Common 包上。
- 对 0.23 新的 HDFS 包做了少量修改使 0.23 的 Mapred 包能运行在新的 HDFS 包。
- 对云梯的 Common 包的通信部分做了 hack，使其兼容 0.23 的 Common。

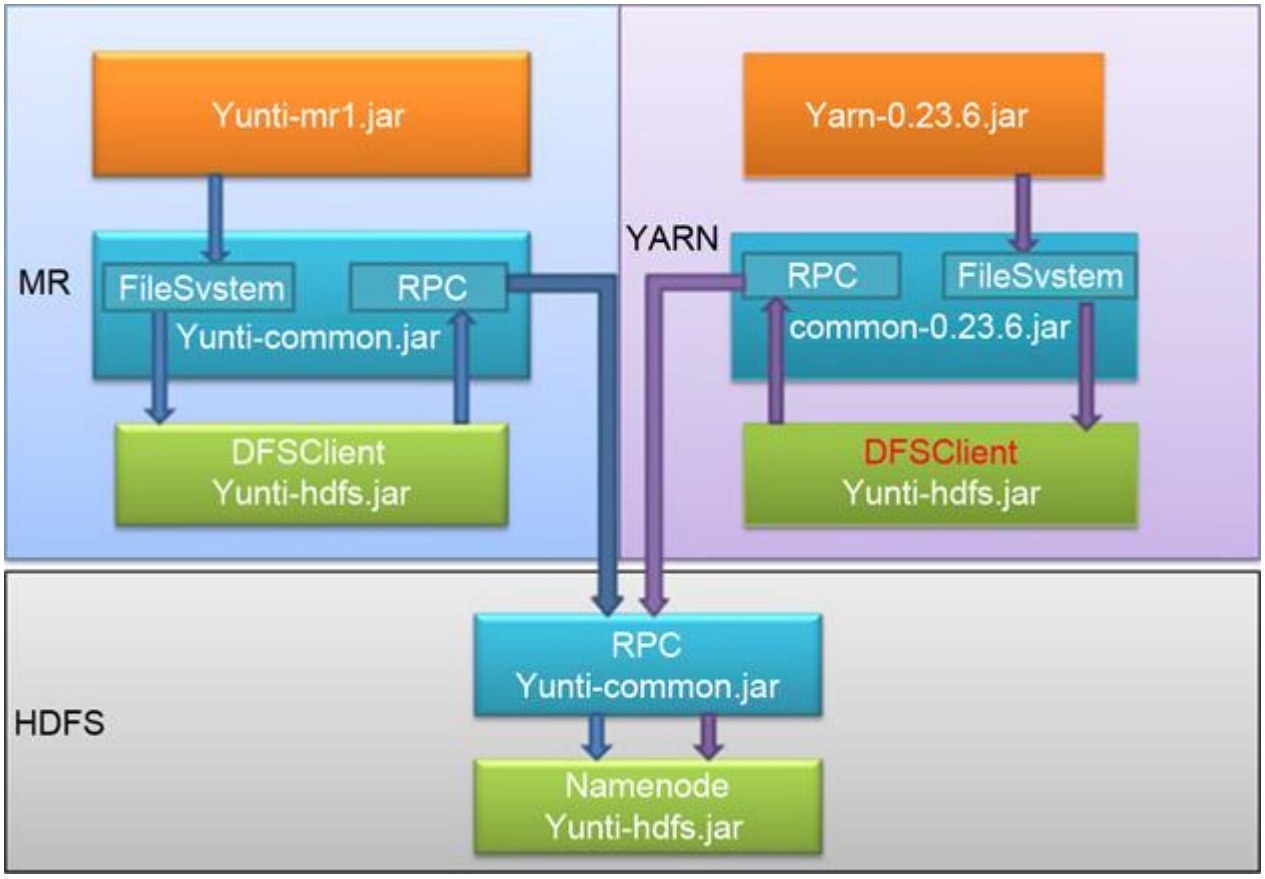


图 1 云梯 Hadoop 代码架构

新的云梯代码结构如图 1 所示，相应阐述如下。

服务端

- 存储部分使用原有的 HDFS。
- MRv1 计算集群中提供原 MRv1 服务。
- YARN 集群提供更丰富的应用服务。

客户端

- 云梯现有的客户端不做任何修改，继续使用原有的服务。

- 使用 YARN 的服务需要使用新客户端。

云梯 MR 服务切换为 YARN 要经过三个阶段

- 服务端只有 MRv1， 客户端只有老版本客户端。
- 服务端 MRv1 和 YARN 共存（MRv1 资源逐渐转移到 YARN 上）， 客户端若需使用 MRv1 服务则保持客户端不变；若需使用 YARN 服务则需使用新版客户端。
- 服务端只剩下 YARN，客户端只有新版本客户端。

通过上述修改，云梯开发人员以较小的修改实现了 YARN 对云梯 HDFS 的访问。

Spark on YARN 的实现

云梯版 YARN 集群已实现对 MRv2、Hive、Spark、MPI、RHive、RHadoop 等应用的支持。云梯集群当前结构如图 2 所示。

图 2 云梯架构图

其中，Spark 已成为 YARN 集群上除 MapReduce 应用外另一个重要的应用。

Spark 是一个分布式数据快速分析项目。它的核心技术是弹性分布式数据集（Resilient Distributed Datasets），提供了比 MapReduce 丰富的模型，可以快速在内存中对数据集进行多次迭代，来支持复杂的数据挖掘算法和图形计算算法。

Spark 的计算调度方式，从 Mesos 到 Standalone，即自建 Spark 计算集群。虽然 Standalone 方式性能与稳定性都得到了提升，但自建集群毕 竟资源较少，并需要从云梯集群复制数据，不能满足数据挖掘与计算团队业务需求。而 Spark on YARN 能让 Spark 计算模型在云梯 YARN 集群上运行，直接读取云梯上的数据，并充分享受云梯 YARN 集群丰富的计算资源。

Spark on YARN 功能理论上从 Spark 0.6.0 版本开始支持，但实际上还远未成熟，经过数据挖掘与计算团队长时间的压力测试，修复了一些相对关键的 Bug，保证 Spark on YARN 的稳定性和正确性。

图 3 展示了 Spark on YARN 的作业执行机制。

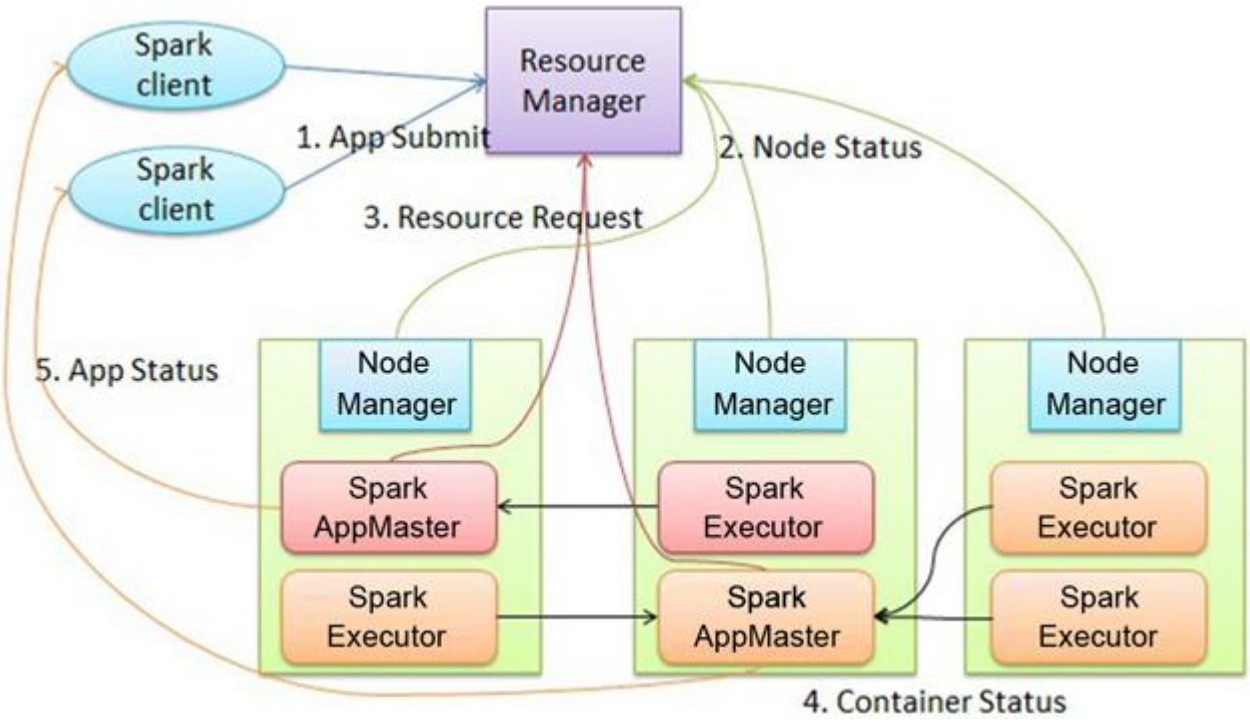


图 3 Spark on YARN 框架

基于 YARN 的 Spark 作业首先由客户端生成作业信息，提交给 ResourceManager，ResourceManager 在某一 NodeManager 汇报时把 AppMaster 分配给 NodeManager,NodeManager 启动 SparkAppMaster，SparkAppMaster 启动后初始化作业，然后向 ResourceManager 申请资源，申请到相应资源后 SparkAppMaster 通过 RPC 让 NodeManager 启动相应的 SparkExecutor，SparkExecutor 向 SparkAppMaster 汇报并完成相应的任务。此外，SparkClient 会通过 AppMaster 获取作业运行状态。

目前，数据挖掘与计算团队通过 Spark on YARN 已实现 MLR、PageRank 和 JMeans 算法，其中 MLR 已作为生产作业运行。

云梯 YARN 集群维护经验分享

云梯 YARN 的维护过程中遇到许多问题，这些问题在维护 YARN 集群中很有可能会遇到，这里分享两个较典型的问题与其解决方法。

- **问题 1**

问题描述：社区的 CPU 隔离与调度功能，需要在每个 NodeManager 所在的机器创建用户账户对应的 Linux 账户。但阿里云梯集群有 5000 多个账 户，是否需要在每个 NodeManager 机器创建这么多 Linux 账户；另外每次创建或删除一个 Hadoop 用户，也应该在每台 NodeManager 机器上创建或删除相应的 Linux 账户，这将大大增加运维的负担。

问题分析：我们发现，CPU 的隔离是不依赖于 Linux 账户的，意味着即 使同一个账户创建两个进程，也可通过 Cgroup 进行 CPU 隔离，但为什么社区要在每台 NodeManager 机器上创建账户呢？原来这是为了让每个 Container 都以提交 Application 的账户执行，防止 Container 所属的 Linux 账户权限过大，保证安全。但云梯集群很早前就已分 账户，启动 Container 的 Linux 账户统一为一个普通账户，此账户权限较小，并且用户都为公司内部员工，安全性已能满足需求。

解决方案：通过修改 container-executor.c 文件，防止其修改 Container 的启动账户，并使用一个统一的普通 Linux 账户（无 sudo 权限）运行 Container。这既能保证安全，又能减少运维的工作量。

- **问题 2**

问题描述：MRApplicationMaster 初始化慢，某些作业的 MRApplicationMaster 启动耗时超过一分钟。

问 题分析：通过检查 MRApplication-Master 的日志，发现一分钟的初始化时间都消耗在解析 Rack 上。从代码上分 析，MRApplicationMaster 启动时需要初始化 TaskAttempt，这时需要解析 split 信息中的 Host，生成对应的 Rack 信 息。云梯当前解析 Host 的方法是通过调用外部一个 Python 脚本解析，每次调用需要 20ms 左右，而由于云梯 HDFS 集群非常大，有 4500 多台机 器，假如输入数据分布在每个 Datanode 上，则解析 Host 需要花费 4500×20ms=90s；如果一个作业的输入数据较大，且文件的备份数为 3， 那么输入数据将很有可能分布在集群的大多 Datanode 上。

解决方案：开发人员通过在 Node-Manager 上增加一个配置文件，包含所有 Datanode 的 Rack 信息，MRApp-licationMaster 启动后加载此文件，防止频繁调用外部脚本解析。这大大加快了 MRApplicationMaster 的初始化速度。

此外，云梯开发人员还解决了一些会使 ResourceManager 不工作的 Bug，并贡献给 Apache Hadoop 社区。

在搭建与维护云梯 YARN 集群期间，云梯开发人员遇到并解决了许多问题，分析和解决这些问题首先需要熟悉代码，但代码量巨大，我们如何能快速熟悉它们呢？这 需要团队的配合，团队中每个人负责不同模块，阅读后轮流分享，这能加快代码熟悉速度。另外，Hadoop 的优势在于可以利用社区的力量，当遇到一个问题 时，首先可以到社区寻找答案，因为很多问题在社区已得到了解决，充分利用社区，可以大大提高工作效率。

云梯 YARN 集群的优势与未来之路

当前云梯 YARN 集群已经试运行，并有 MRv2、Hive、Spark、RHive 和 RHadoop 等应用。云梯 YARN 集群的优势在于：

- 支持更丰富的计算模型；
- 共享云梯最大的存储集群，访问便捷、快速；
- AppHistory 信息存储在 HDFS 上，各种应用的作业历史都能方便查看；
- 相对于 MRv1 集群，云梯 YARN 能支持更大规模的集群；
- 相对于 MRv1 集群，云梯 YARN 集群支持内存和 CPU 调度，资源利用将更加合理。

未来，云梯将会把大多业务迁移到云梯 YARN 集群。针对 YARN 版本，云梯将增加资源隔离与调度，增加对 Storm、Tez 等计算模型的支持，并优化 YARN 的性能。

作者沈洪，花名俞灵，就职于阿里巴巴集团数据平台事业部海量数据部门，目前从事 YARN、MapReduce 的研究、开发与集群的维护。

原文

http://www.csdn.net/article/2013-12-04/2817706?utm_source=Tuicool_Weekly

高并发、大流量网卡调优

周五晚上 LVS 集群的一台 LB 由于 CPU 单核耗尽挂掉了，最后查到原因是网卡调优没有生效，今天查了一下网卡调优的资料，感谢同事 higkoo 给我讲解了一下这方面的东西，下面是关于我们公司网卡调优的相关知识，欢迎大家共同探讨，让我们的机器跑的更 high。

1、Broadcom 的网卡建议关闭 GRO 功能

```
ethtool -K eth0 gro off

ethtool -K eth1 gro off

ethtool -K eth2 gro off

ethtool -K eth3 gro off
```

2、关闭 irqbalance 服务并手动分配网卡中断

```
service irqbalance stop

chkconfig irqbalance off
```

查看网卡中断号

grep ethx /proc/interrupts

分配到每颗颗CPU核上

cat /proc/irq/{84,85,86,87,88,89,90,91,92,93}/smp_affinity （下面echo的值从此获取）

echo 1 > /proc/irq/84/smp_affinity

echo 2 > /proc/irq/85/smp_affinity

echo 4 > /proc/irq/86/smp_affinity

echo 8 > /proc/irq/87/smp_affinity

echo 10 > /proc/irq/88/smp_affinity

echo 20 > /proc/irq/89/smp_affinity

echo 40 > /proc/irq/90/smp_affinity

echo 80 > /proc/irq/91/smp_affinity

echo 100 > /proc/irq/92/smp_affinity

echo 200 > /proc/irq/93/smp_affinity

3、开启网卡的RPS功能 （Linux内核2.6.38或以上版本支持）

Enable RPS (Receive Packet Steering)

rfc=4096

cc=$(grep -c processor /proc/cpuinfo)

rsfe=$(echo $cc*$rfc | bc)

sysctl -w net.core.rps_sock_flow_entries=$rsfe

```
for fileRps in $(ls /sys/class/net/eth*/queues/rx-*/rps_cpus)

do

echo fff > $fileRps

done

for fileRfc in $(ls /sys/class/net/eth*/queues/rx-*/rps_flow_cnt)

do

echo $rfc > $fileRfc

done

tail /sys/class/net/eth*/queues/rx-*/{rps_cpus,rps_flow_cnt}
```

献上一个完整的脚本：

```
vi /opt/sbin/change_irq.sh

#!/bin/bash

ethtool -K eth0 gro off

ethtool -K eth1 gro off

ethtool -K eth2 gro off

ethtool -K eth3 gro off

service irqbalance stop

chkconfig irqbalance off

cat /proc/irq/{84,85,86,87,88,89,90,91,92,93}/smp_affinity

echo 1 > /proc/irq/84/smp_affinity

echo 2 > /proc/irq/85/smp_affinity

echo 4 > /proc/irq/86/smp_affinity
```

```
echo 8 > /proc/irq/87/smp_affinity

echo 10 > /proc/irq/88/smp_affinity

echo 20 > /proc/irq/89/smp_affinity

echo 40 > /proc/irq/90/smp_affinity

echo 80 > /proc/irq/91/smp_affinity

echo 100 > /proc/irq/92/smp_affinity

echo 200 > /proc/irq/93/smp_affinity

# Enable RPS (Receive Packet Steering)

rfc=4096

cc=$(grep -c processor /proc/cpuinfo)

rsfe=$(echo $cc*$rfc | bc)

sysctl -w net.core.rps_sock_flow_entries=$rsfe

for fileRps in $(ls /sys/class/net/eth*/queues/rx-*/rps_cpus)

do

echo fff > $fileRps

done

for fileRfc in $(ls /sys/class/net/eth*/queues/rx-*/rps_flow_cnt)

do

echo $rfc > $fileRfc

done

tail /sys/class/net/eth*/queues/rx-*/{rps_cpus,rps_flow_cnt}

chmod +x /opt/sbin/change_irq.sh
```

echo "/opt/sbin/change_irq.sh" >> /etc/rc.local

PS：记得修改网卡中断号，别直接拿来用哦

原文
http://navyaijm.blog.51cto.com/4647068/1334671?utm_source=Tuicool_Weekly

Impala：新一代开源大数据分析引擎

摘要：大数据处理是云计算中非常重要的领域，自 Google 公司提出 MapReduce 分布式处理框架以来，以 Hadoop 为代表的开源软件受到越来越多公司的重视和青睐。本文将讲述 Hadoop 系统中的一个新成员：Impala。

Impala 架构分析

Impala 是 Cloudera 公司主导开发的新型查询系统，它提供 SQL 语义，能查询存储在 Hadoop 的 HDFS 和 HBase 中的 PB 级大数据。已有的 Hive 系统虽然也提供了 SQL 语义，但由于 Hive 底层执行使用的是 MapReduce 引擎，仍然是一个批处理过程，难以满足查询的交互性。相比之下，Impala 的最大特点也是最大卖点就是它的快速。那么 Impala 如何实现大数据的快速查询呢？在回答这个问题前，需要先介绍 Google 的 Dremel 系统，因为 Impala 最开始是参照 Dremel 系统进行设计的。

Dremel 是 Google 的交互式数据分析系统，它构建于 Google 的 GFS（Google File System）等系统之上，支撑了 Google 的数据分析服务 BigQuery 等诸多服务。Dremel 的技术亮点主要有两个：一是实现了嵌套型数据的列存储；二是使用了多层查询树，使得任务可以在数千个节点上并行执行和聚合结果。列存储在关系型数据库中并不陌生，它可以减少查询时处理的数据量，有效提升 查询效率。Dremel 的列存储的不同之处在于它针对的并不是传统的关系数据，而是嵌套结构的数据。Dremel 可以将一条条的嵌套结构的记录转换成列存 储形式，查询时根据查询条件读取需要的列，然后进行条件过滤，输出时再将列组装成嵌套结构的记录输出，记录的正向和反向转换都通过高效的状态机实现。另 外，Dremel 的多层查询树则借鉴了分布式搜索引擎的设计，查询树的根节点负责接收查询，并将查询分发到下一层节点，底层节点负责具体的数据读取和查询 执行，然后将结果返回上层节点。关于 Dremel 技术实现上的更多信息，可以参阅【注：Google Dremel 原理：如何能 3 秒分析 1PB，网址为 http://www.yankay.com/google-dremel-rationale/】。

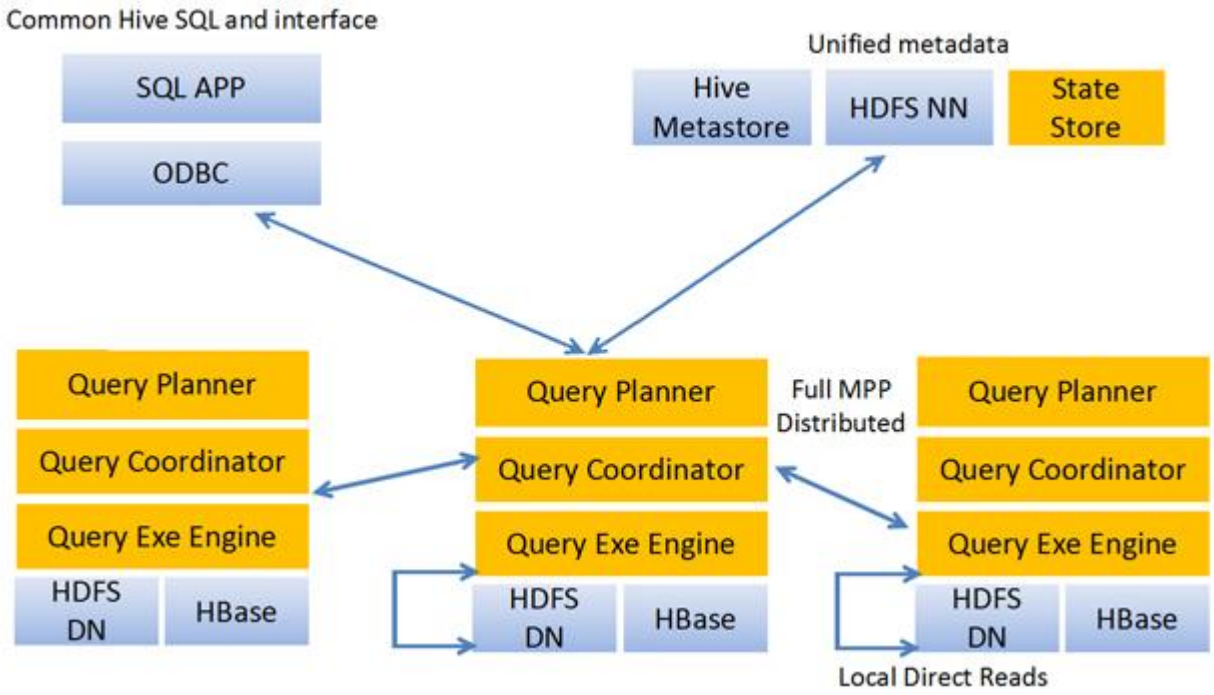


图 1 Impala 系统架构图

Impala 其实就是 Hadoop 的 Dremel，Impala 使用的列存储格式是 Parquet。Parquet 实现了 Dremel 中的列存储，未来还将支持 Hive 并添加字典编码、游程编码等功能。Impala 的系统架构如图 1 所示。Impala 使用了 Hive 的 SQL 接口(包括 SELECT、 INSERT、Join 等操作)，但目前只实现了 Hive 的 SQL 语义的子集（例如尚未对 UDF 提供支持），表的元数据信息存储在 Hive 的 Metastore 中。StateStore 是 Impala 的一个子服务，用来监控集群中各个节点的健康状况，提供节点注册、错误检测等功能。 Impala 在每个节点运行了一个后台服务 Impalad，Impalad 用来响应外部请求，并完成实际的查询处理。Impalad 主要包含 Query Planner、Query Coordinator 和 Query Exec Engine 三个模块。QueryPalnner 接收来自 SQL APP 和 ODBC 的查询，然后将查询转换为许多子查询，Query Coordinator 将这些子查询分发到各个节点上，由各个节点上的 Query Exec Engine 负责子查询的执行，最后返回子查询的结果，这些中间结果经过聚集之后最终返回给用户。

在 Cloudera 的测试中，Impala 的查询效率比 Hive 有数量级的提升。从技术角度上来看，Impala 之所以能有好的性能，主要有以下几方面的原因。

- Impala 不需要把中间结果写入磁盘，省掉了大量的 I/O 开销。

- 省掉了 MapReduce 作业启动的开销。MapReduce 启动 task 的速度很慢(默认每个心跳间隔是 3 秒钟)，Impala 直接通过相应的服务进程来进行作业调度，速度快了很多。
- Impala 完全抛弃了 MapReduce 这个不太适合做 SQL 查询的范式，而是像 Dremel 一样借鉴了 MPP 并行数据库的思想另起炉灶，因此可做更多的查询优化，从而省掉不必要的 shuffle、sort 等开销。
- 通过使用 LLVM 来统一编译运行时代码，避免了为支持通用编译而带来的不必要开销。
- 用 C++实现，做了很多有针对性的硬件优化，例如使用 SSE 指令。
- 使用了支持 Data locality 的 I/O 调度机制，尽可能地将数据和计算分配在同一台机器上进行，减少了网络开销。

虽然 Impala 是参照 Dremel 来实现的，但它也有一些自己的特色，例如 Impala 不仅支持 Parquet 格式，同时也可以直接处理文本、 SequenceFile 等 Hadoop 中常用的文件格式。另外一个更关键的地方在于，Impala 是开源的，再加上 Cloudera 在 Hadoop 领域 的领导地位，其生态圈有很大可能会在将来快速成长。

可以预见,在不久的未来,Impala 很可能像之前的 Hadoop 和 Hive 一样在大数 据处理领域大展拳脚。Cloudera 自己也说期待未来 Impala 能完全取代 Hive。当然，用户从 Hive 上迁移到 Impala 上来是需要时间的,而 且 Impala 也只是刚刚发布 1.0 版，虽然号称已经可以稳定地在生产环境上运行，但相信仍然有很多可改进的空间。需要说明的是，Impala 并不是用来 取代已有的 MapReduce 系统，而是作为 MapReduce 的一个强力补充。总的来说，Impala 适合用来处理输出数据适中或比较小的查询，而对于 大数据量的批处理任务，MapReduce 依然是更好的选择。另外一个消息是，Cloudera 里负责 Impala 的架构师 Marcel Komacker 就曾在 Google 负责过 F1 系统的查询引擎开发，可见 Google 确实为大数据的流行出钱出力。

Impala 与 Shark、Drill 等的比较

开源组织 Apache 也发起了名为 Drill 的项目来实现 Hadoop 上的 Dremel，目前该项目正在开发中，相关的文档和代码还不多，暂时还未对 Impala 构成足够的威胁【注：Isn't Cloudera Impala doing the same job as Apache Drill incubator project? ，网址为：
http://www.quora.com/Cloudera-Impala/Isnt-Cloudera-Impala-doing-the-same-job-as-Apache-Drill-incubator-project】。

从 Quora 上的问答来 看，Cloudera 有 7~8 名工程师全职在 Impala 项目上，而相比之下 Drill 目前的动作稍显迟钝。具体来说，截至 2012 年 10 月 底，Drill

的代码库里实现了 query parser、plan parser 及能对 JSON 格式的数据进行扫描的 plan evaluator；而 Impala 同期已经有了一个比较完备的分布式 query execution 引擎，并对 HDFS 和 HBase 上的数据读入、错误检测、INSERT 的数据修改、LLVM 动态翻译等都提供了支持。当然，Drill 作为 Apache 的项目，从一开始就避免了某个 vendor 的一家独大，而且对所有 Hadoop 流行的发行版都会做相应的支持，不像 Impala 只支持 Cloudera 自己的发行版 CDH。从长远来看，谁会占据上风还真不一定。

除此之外，加州伯克利大学 AMPLab 也开发了名为 Shark 的大数据分析系统。本刊 7 月期《Spark：大数据时代的“电光石火”》一文专门分析了与 Shark 相关的 Spark 系统，感兴趣的读者朋友可以参考。从 长远目标来看，Shark 想成为一个既支持大数据 SQL 查询，又能支持高级数据分析任务的一体化数据处理系统。从技术实现的角度上来看，Shark 基于 Scala 语言的算子推导实现了良好的容错机制，因此对失败了的长任务和短任务都能从上一个“快照点”进行快速恢复。

相比之下，Impala 由于缺失足够强大的容错机制，其上运行的任务一旦失败就必须“从头来过”，这样的设计必然会在性能上有所缺失。而且 Shark 是把内存当作第一类的存储介质来做的系统设计，所以在处理速度上也会有一些优势。

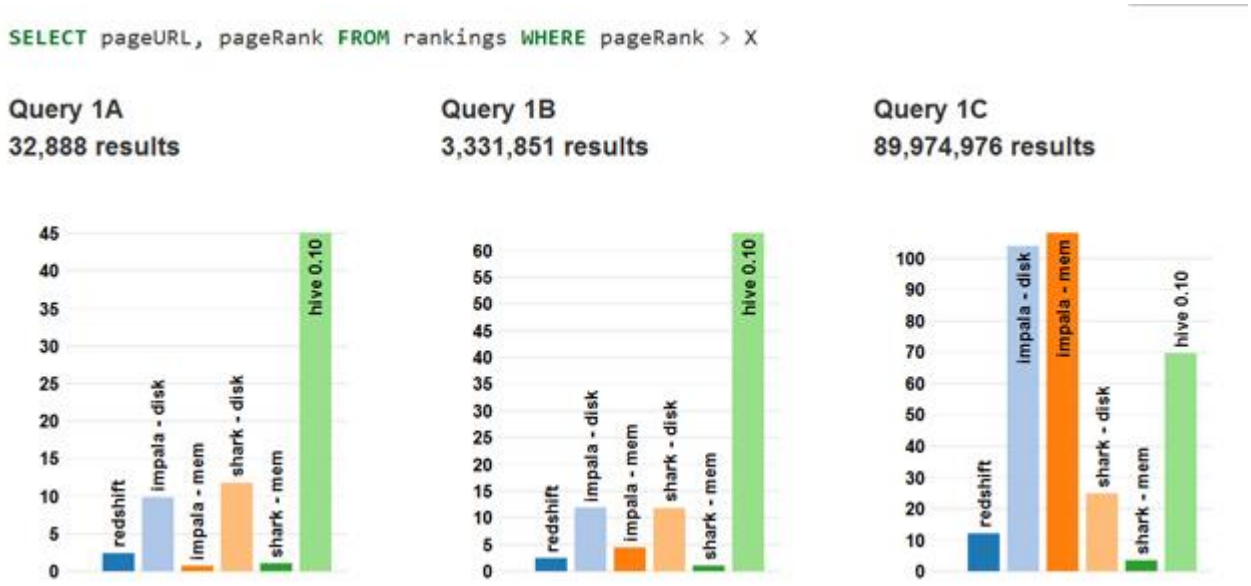


图 2 Redshift、Impala、Shark 与 Hive 的 Aggregation Query 性能对比

实际上，AMPLab 最近对 Hive、Impala、Shark 及 Amazon 采用的商业 MPP 数据库 Redshift 进行了一次对比试验，在 Scan Query、Aggregation Query 和 Join Query 三种类型的任务中对它们进行了比较。图 2 就是 AMPLab 报告中 Aggregation Query 的性能对比。在图中我们可以看到，商业版本的 Redshift 的性能是最好的， Impala 和 Shark 则各有胜负，且两者都比 Hive 的性能高出了一大截。更多相关的实验结果读者朋友可以参考【注：Big Data Benchmark:https://amplab.cs.berkeley.edu/benchmark/】。

其实对大数据分析的项目来说，技术往往不是最关键的。例如 Hadoop 中的 MapReduce 和 HDFS 都是源于 Google，原创性较少。事实上，开源项目的生态圈、社区、发展速度等，往往在很大程度上会影响 Impala 和 Shark 等开源大数据分析系统的发展。就像 Cloudera 一开始就决定会把 Impala 开源，以期望利用开源社区的力量来推广这个产品；Shark 也是一开始就开源了出来，更不用说 Apache 的 Drill 更是如此。说到底还是 谁的生态系统更强的问题。技术上一时的领先并不足以保证项目的最终成功。虽然最后哪一款产品会成为事实上的标准还很难说，但我们唯一可以确定并坚信的一点 是，大数据分析将随着新技术的不断推陈出新而不断普及开来，这对用户永远都是一件幸事。

举个例子，如果读者注意过下一代 Hadoop（YARN）的发展的话就会发现，其实 YARN 已经支持 MapReduce 之外的计算范式（例如 Shark、Impala 等），因此将来 Hadoop 将可能作为一个兼容并包的大平台存在，在其上提供各种各样的数据处理技术，有应对秒量级查询的，有应对大数据批处理的，各种功能应有尽有，满 足用户各方面的需求。

未来展望

其实除了 Impala、 Shark 和 Drill 这样的开源方案外，像 Oracle、EMC 等传统厂商也没在坐以待毙等着自己的市场被开源软件侵吞。像 EMC 就推出了 HAWQ 系统，并号称其性能比之 Impala 快上十几倍，而前面提到的 Amazon 的 Redshift 也提供了比 Impala 更好的性能。

虽然说开源软件因为其强大的成本优势而拥有极其强大的力量，但传统数据库厂商仍会尝试推出性能、稳定性、维护服务等指标上更加强大的产品与之进行差异化竞争，并同时参与开源社区、借力开源软件来丰富自己的产品线、提升自己的竞争力，并通过更多的高附加值服务来满足某些消费者需求。毕竟，这些厂商往往已在并行数据库等 传统领域积累了大量的技术和经验，底蕴非常深厚。甚至现在还有像 NuoDB（一个创业公司）这样号称既支持 ACID，又有 Scalability 的 NewSQL 系统涌现出来。

总的来看，未来的大数据分析技术将会变得越来越成熟、越来越便宜、越来越易用。与此相应的，用户将会更容易、更方便地从自己的大数据中挖掘出有价值的商业信息。

原文
http://www.csdn.net/article/2013-12-04/2817707?utm_source=Tuicool_Weekly

Redis+Keepalived 高可用方案详细分析

上一篇简单的了解了一下 redis 官方自带的 HA 方案 sentinel，试用发现还是不错的，但是由于还没有合并进稳定分支中，所以在生产环境也不敢使用，还有一个就是需求还暂时不能完全满足，所以就尝试一下 redis+keepalived 方案，毕竟 keepalived 现在还是很稳定的，而且资料也充足。

实验环境

ubuntu12.04　10.20.112.26 默认的 master
　　10.20.112.27 默认的 slave
VIP　　10.20.112.29
redis-server　2.6.16

keepalived 默认只能做到对网络故障和 keepalived 本身的监控，即当出现网络故障或者 keepalived 本身出现问题时，进行切换。但我们更关注的是机器上运行的业务，如果业务出问题了 VIP 没有变化，整体来说还是失败的。这时候就需要根据业务进程的运行状态决定是否需要进行主备切换。还好 keepalived 提供了这样一个自定义脚本监控功能，不过该参数设置在官方默认的文档中并没有出现。

其实文档中有两个我们常用的参数都没有提到：

1　vrrp_script && track_script

vrrp_script 代码块是用来定义监控脚本，脚本执行时间间隔以及脚本的执行结果导致优先级变更幅度的。

```
1 vrrp_script chk_redis {
        script  "/etc/keepalived/scripts/redis_check.sh"     #指定执
2 行脚本的路径
        interval  1
3                                #指定脚本的执行时间间隔
        weight  10
4                                #脚本结果导致的优先级变更：10 表示优先
```

级+10；-10 则表示优先级-10
5}

定义好 vrrp_script 代码块之后，就可以在 instance 中使用了

```
1 track_script {
2           chk_redis
3}
```

注意：VRRP 脚本(vrrp_script)和 VRRP 实例(vrrp_instance)属于同一个级别

2 notify_stop

keepalived 停止运行前运行 notify_stop 指定的脚本。

配合官方文档提到的以下三个参数一起使用，功能更强大：

notify_master keepalived 切换到 master 时执行的脚本

notify_backup keepalived 切换到 backup 时执行的脚本

notify_fault keepalived 出现故障时执行的脚本

3 还有个问题需要注意

当 master down 了，backup 接管了，master 再次起来，不能再成为 master。否则 master 恢复了再接管的话，会造成业务来回切换，这时候就需要 nopreempt 参数了。

nopreempt：设置不抢占，这里只能设置在 state 为 backup 的节点上，而且这个节点的优先级必须别另外的高。

先来看看方案的整体思路：

通过 keepalived 的自定义脚本功能监控本机的 redis 服务状态，当监控脚本检测到 redis 服务出现异常时,则改变本机 keepalived 的优先级，同时这会导致 master/backup 角色的变化，而 keepalived 在角色变化时也会触发一些机制执行相关脚本，这就为我们改变 redis 的 master/slave 状态提供了机会，这样做的目的是为了是 redis 的 master/slave 直接的数据保持一致。

在 keepalived+redis 的使用过程中有四种情况：

1 一种是 keepalived 挂了，同时 redis 也挂了，这样的话直接 VIP 飘走之后，是不需要进行 redis 数据同步的，因为 redis 挂了，你也无法去 master 上同步，不过会损失已经写在 master 上却还没同步到 slave 上面的这部分数据。

2 另一种是 keepalived 挂了,redis 没挂,这时候 VIP 飘走后,redis 的 master/slave 还是老的对应关系，如果不变化的话会把数据写入 redis slave 中，从而不会同步到 master 上去，这就要借助监控脚本反转 redis 的 master/slave 关系。这时候就要预留一点时间进行数据同步，然后反转 master/slave。

3 还有一种是 keepalived 没挂，redis 挂了，这时候根据监控脚本会检测到 redis 挂了，并且降低 keepalived master 的优先级，同样会导致 VIP 飘走，情况和第二种一样，也是需要进行数据同步，然后反转当前 redis 的 master/slave 关系的。

4 随后一种是 keepalived 没挂，redis 也没挂，大吉大利啊，什么都不用操作。

本文的实验环境四种情况都适合，第一种是不需要同步数据的，脚本会默认去同步数据，但是其实是不会成功的。脚本主要是用来处理第二和第三种情况的。

配置 10.20.112.26

1 安装 keepalived

```
1 apt-get install keepalived
```

2 安装 redis

redis 是采用源码编译安装的，ubuntu12.04 默认自带版本没有这么高，安装过程参照以前文档。

3 配置 keepalived

```
01 global_defs {
02         lvs_id LVS_redis
03 }
04 vrrp_script chk_redis {
05                 script  "/etc/keepalived/scripts/redis_check.sh"
06                 weight  -20
0                 interval  2
7
08          }
09
10 vrrp_instance VI_1 {
                   state
11 backup
                   interface
12 eth0
13                 virtual_router_id  51
14                 nopreempt
1
5                  priority  200
1                  advert_int  5
```

```
6
17                  track_script {
1                           chk_redis
8
19                  }
20                  virtual_ipaddress {
2                             10.20.112.29
1
22                  }
2                   notify_master  /etc/keepalived/scripts/redis_maste
3 r.sh
2                   notify_backup  /etc/keepalived/scripts/redis_backu
4 p.sh
2                   notify_fault    /etc/keepalived/scripts/redis_faul
5 t.sh
2                   notify_stop      /etc/keepalived/scripts/redis_sto
6 p.sh
27}
```

4 配置 redis，/etc/redis/redis.com

最简单的配置就是把默认配置文件中的 bind 127.0.0.1 修改为 0.0.0.0 即可。

5 建立 redis 状态切换脚本

在/etc/keepalived 目录下建立 log 和 scripts 目录。

在 script 下有五个脚本，一个是检测 redis 状态的 redis_check.sh 脚本，其余四个是 keepalived 状态变化时执行的脚本。keepalived 有 master/backup/stop/fault 四种状态，因为我们主要是关注系统上的业务，所以在在 keepalived 进入 fault/stop 状态后，也认为是进入了 backup 状态，需要对 redis 的 master/slave 关系进行反转，否则即使 VIP 漂移过去，但是 redis 的主从关系还没有改变，会导致数据不一致，所以最终四个脚本只有两种内容。

5.1 redis 服务状态检测脚本 redis_check.sh(27 上面内容和它一样)

```
01 #!/bin/bash
02 ###/etc/keepalived/scripts/redis_check.sh
03 ALIVE=`/usr/bin/redis-cli PING`
04 if  [  "$ALIVE"  ==  "PONG"  ]; then
05     echo $ALIVE
06     exit  0
07 else
```

```
08     echo $ALIVE
09     exit  1
10 fi
```

5.2 keepalived 进入 master 状态时的检测脚本 redis_master.sh

```
01 #!/bin/bash
02 ###/etc/keepalived/scripts/redis_master.sh
03 REDISCLI="redis-cli"
04 LOGFILE="/etc/keepalived/log/redis-state.log"
05 pid=$$
06
07 echo  "`date        +'%Y-%m-%d:%H:%M:%S'`|$pid|state:[slaver]"  >> $LOGFILE
08 echo  "`date +'%Y-%m-%d:%H:%M:%S'`|$pid|state:[slaver] Run 'SLAVEOF 10.20.112.27 6379'"  >> $LOGFILE
09 $REDISCLI SLAVEOF  10.20.112.27  6379  >> $LOGFILE    2>&1
10 echo  "`date +'%Y-%m-%d:%H:%M:%S'`|$pid|state:[slaver] wait 10 sec for data sync from old master"  >> $LOGFILE
11 sleep  10
12 echo  "`date  +'%Y-%m-%d:%H:%M:%S'`|$pid|state:[slaver]  data  rsync from old mater ok..."  >> $LOGFILE
13 echo  "`date +'%Y-%m-%d:%H:%M:%S'`|$pid|state:[master] Run slaveof no one,close master/slave"  >> $LOGFILE
14 $REDISCLI SLAVEOF NO ONE >> $LOGFILE  2>&1
15 echo  "`date  +'%Y-%m-%d:%H:%M:%S'`|$pid|state:[master]  wait  other slave connect...."  >> $LOGFILE
```

5.3 keepalived 进入 backup/stop/fault 时的检测脚本，由于内容都一致，所以只写出 redis_backup.sh

```
01 #!/bin/bash
02 ###/etc/keepalived/scripts/redis_backup.sh
03 REDISCLI="redis-cli"
04 LOGFILE="/etc/keepalived/log/redis-state.log"
05 pid=$$
06
07 echo  "`date        +'%Y-%m-%d:%H:%M:%S'`|$pid|state:[master]"  >> $LOGFILE
08 echo  "`date +'%Y-%m-%d:%H:%M:%S'`|$pid|state:[master] Being slave state..."  >> $LOGFILE  2>&1
```

```
09 echo "`date +'%Y-%m-%d:%H:%M:%S'`|$pid|state:[master] wait 10 sec for data sync from old master" >> $LOGFILE
10 sleep 10
11 echo "`date +'%Y-%m-%d:%H:%M:%S'`|$pid|state:[master] data rsync from old mater ok..." >> $LOGFILE
12 echo "`date +'%Y-%m-%d:%H:%M:%S'`|$pid|state:[slaver] Run 'SLAVEOF 10.20.112.27 6379'" >> $LOGFILE
13 $REDISCLI SLAVEOF 10.20.112.27 6379 >> $LOGFILE 2>&1
14 echo "`date +'%Y-%m-%d:%H:%M:%S'`|$pid|state:[slaver] slave connect to 10.20.112.27 ok..." >> $LOGFILE
```

配置 10.20.112.27

1 安装 keepalived

2 安装 redis

3 配置 redis，/etc/redis/redis.com

前面三步骤均一样

4 配置 keepalived

```
01 global_defs {
02         lvs_id LVS_redis
03 }
04
05 vrrp_script chk_redis {
06                 script "/etc/keepalived/scripts/redis_check.sh"
07                 weight -20
08                 interval 2
09 }
10 vrrp_instance VI_1 {
11                 state backup
12                 interface eth0
13                 virtual_router_id 51
14                 priority 190
15                 advert_int 5
```

```
5
16                  track_script {
1                           chk_redis
7
1
8                   }
19                  virtual_ipaddress {
2                           10.20.112.29
0
2
1                   }
2                   notify_master  /etc/keepalived/scripts/redis_maste
2 r.sh
2                   notify_backup  /etc/keepalived/scripts/redis_backu
3 p.sh
2                   notify_fault    /etc/keepalived/scripts/redis_faul
4 t.sh
2                   notify_stop      /etc/keepalived/scripts/redis_sto
5 p.sh
2
6 }
```

5 建立 redis 状态切换脚本

在/etc/keepalived 目录下建立 log 和 scripts 目录

5.1 redis 服务状态检测脚本 redis_check.sh(26 上面内容和它一样)

5.2 keepalived 进入 master 状态时的检测脚本 redis_master.sh

```
01 #!/bin/bash
02 ###/etc/keepalived/scripts/redis_master.sh
03 REDISCLI="/usr/bin/redis-cli"
04 LOGFILE="/etc/keepalived/log/redis-state.log"
05 pid=$$
06
07 echo  "`date        +'%Y-%m-%d:%H:%M:%S'`|$pid|state:[backup]"  >>
   $LOGFILE
08 echo  "`date +'%Y-%m-%d:%H:%M:%S'`|$pid|state:[backup] Run 'SLAVEOF
   10.20.112.26 6379'"  >> $LOGFILE
09 $REDISCLI SLAVEOF  10.20.112.26  6379  >> $LOGFILE    2>&1
10 echo  "`date +'%Y-%m-%d:%H:%M:%S'`|$pid|state:[backup] wait 10 sec
```

```
for data sync from old master"  >> $LOGFILE
11 sleep  10
12 echo  "`date  +'%Y-%m-%d:%H:%M:%S'`|$pid|state:[master]  data  rsync from old mater ok..."  >> $LOGFILE
13 echo  "`date  +'%Y-%m-%d:%H:%M:%S'`|$pid|state:[master]  Run  slaveof no one,close master/slave"  >> $LOGFILE
14 $REDISCLI SLAVEOF NO ONE >> $LOGFILE  2>&1
15 echo  "`date  +'%Y-%m-%d:%H:%M:%S'`|$pid|state:[master]  wait  other slave connect...."  >> $LOGFILE
```

5.3 keepalived 进入 backup/stop/fault 时的检测脚本，由于内容都一致，所以只写出 redis_backup.sh

```
01 #!/bin/bash
02 ###/etc/keepalived/scripts/redis_backup.sh
03 REDISCLI="/usr/bin/redis-cli"
04 LOGFILE="/etc/keepalived/log/redis-state.log"
05 pid=$$
06
07 echo  "`date  +'%Y-%m-%d:%H:%M:%S'`|$pid|state:[master]  Being  slave state..."  >> $LOGFILE  2>&1
08 echo  "`date  +'%Y-%m-%d:%H:%M:%S'`|$pid|state:[master]  wait  15  sec for data sync from old master"  >> $LOGFILE
09 sleep  15
10 echo  "`date  +'%Y-%m-%d:%H:%M:%S'`|$pid|state:[master]  data  rsync from old mater ok..."  >> $LOGFILE
11 echo  "`date +'%Y-%m-%d:%H:%M:%S'`|$pid|state:[slaver] Run 'SLAVEOF 10.20.112.26 6379'"  >> $LOGFILE
12 $REDISCLI SLAVEOF  10.20.112.26  6379  >> $LOGFILE    2>&1
13 echo  "`date +'%Y-%m-%d:%H:%M:%S'`|$pid|state:[slaver] slave connect to 10.20.112.26 ok..."  >> $LOGFILE
```

下面开始试验

既然我们设置了 nopreempt，那么在启动 keepalived 的时候就有启动的顺序问题了，我们把 redis 的 master 和 keepalived 的 master(虽然配置文件中都是 backup，但是我们是想让 26 这台做 master 的)默认设置在同一台机器上，由于在 keepalived 的 master 上面设置了 nopreempt 参数，所以在启动 keepalived 服务的时候，一定要先启动 redis master 的那台，因为在设置了 nopreempt 了，keepalived 在启动后都是先进入 backup 状态，

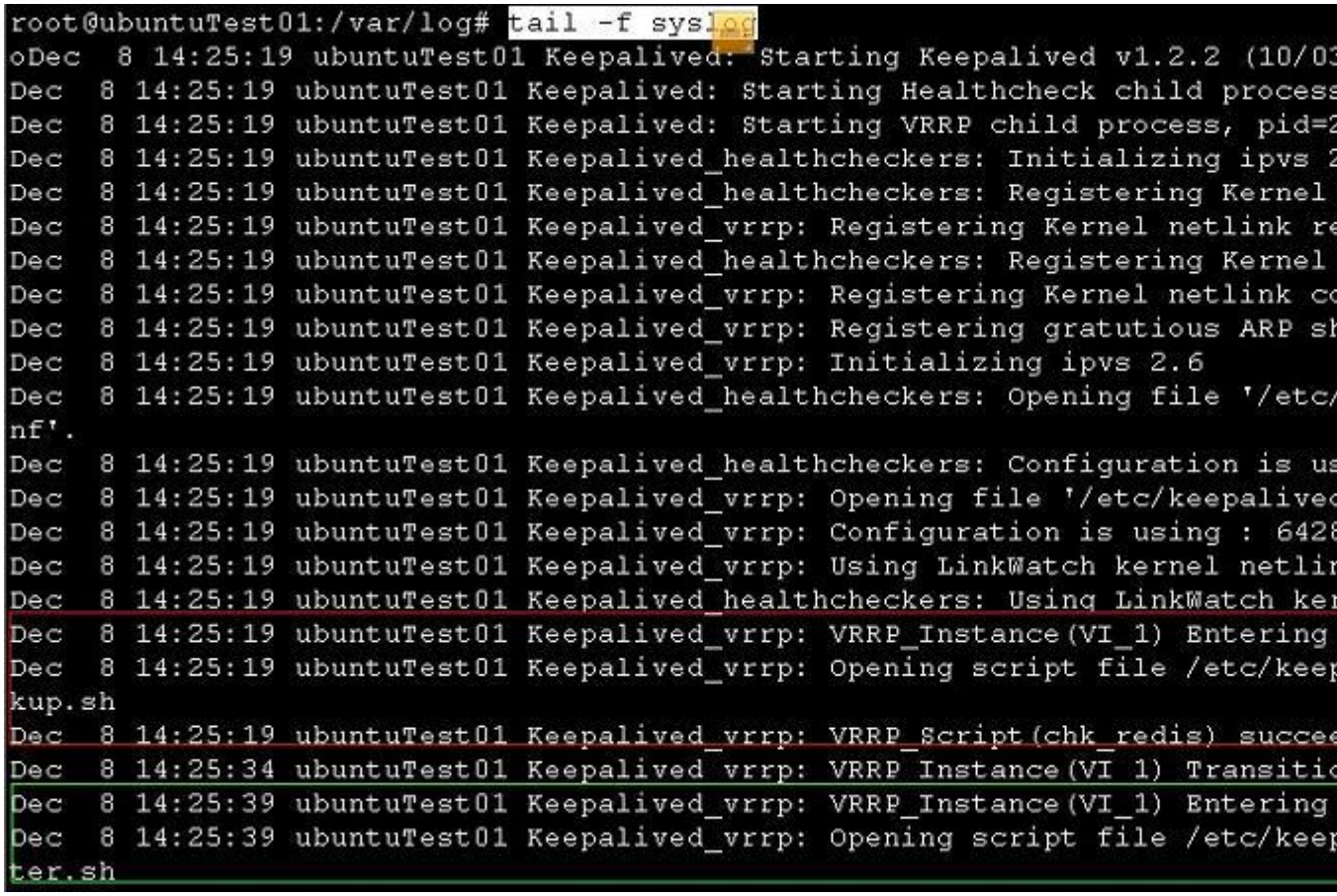
而脚本又设置了进入 backup 状态后，会连接新的对方进行数据同步，所以，在启动 keepalived 之前还有一个条件就是 redis 的 master 和 slave 中的数据必须一致。这样先启动 redis 的 master 那台的 keepalived,虽然 redis master 会连接到 redis slave 同步数据，但是两边数据在刚开始的时候是一致的，并不会产生什么问题。

1 先启动 26 和 27 上的 redis 服务，配置 27 从 26 上面同步数据，同步完毕后，取消 27 的同步机制。

这个就是在 27 的 redis 上面执行 slaveof 10.20.112.26 6379，等待数据同步完毕后再执行 slaveos on one，让 26 和 27 的 redis 都保持 master 状态

2 接着启动 26 的 keepalived，不要启动 27 的 keepalived，在 26 和 27 上面各开启三个终端，观察自定义日志和系统日志状态

26 的 syslog



红框　19 秒 keepalived 启动后，由于设置了不抢占，且起始状态都是 backup，所以启动后 keepalived 先进入 backup 状态，进入 bankup 状态后，成功的执行了 redis_backup.sh 脚本

绿框　34 秒的时候，keepalived 开始向 master 状态转变，注意：这个时间很重要，为什么是 34 秒呢？默认是 3 秒后就开始向 master 状态转变，由于我们设置了 advert_int 5，默认检测三次，所以 15 秒之后才向

master 转变，为什么设置间隔这么长时间呢？是为了让 redis_backup.sh 脚本有充足的时间执行完毕。39 秒的时候成功的转变为 master 状态，这时候会执行 redis_master.sh 脚本。

26 的 redis-state.log

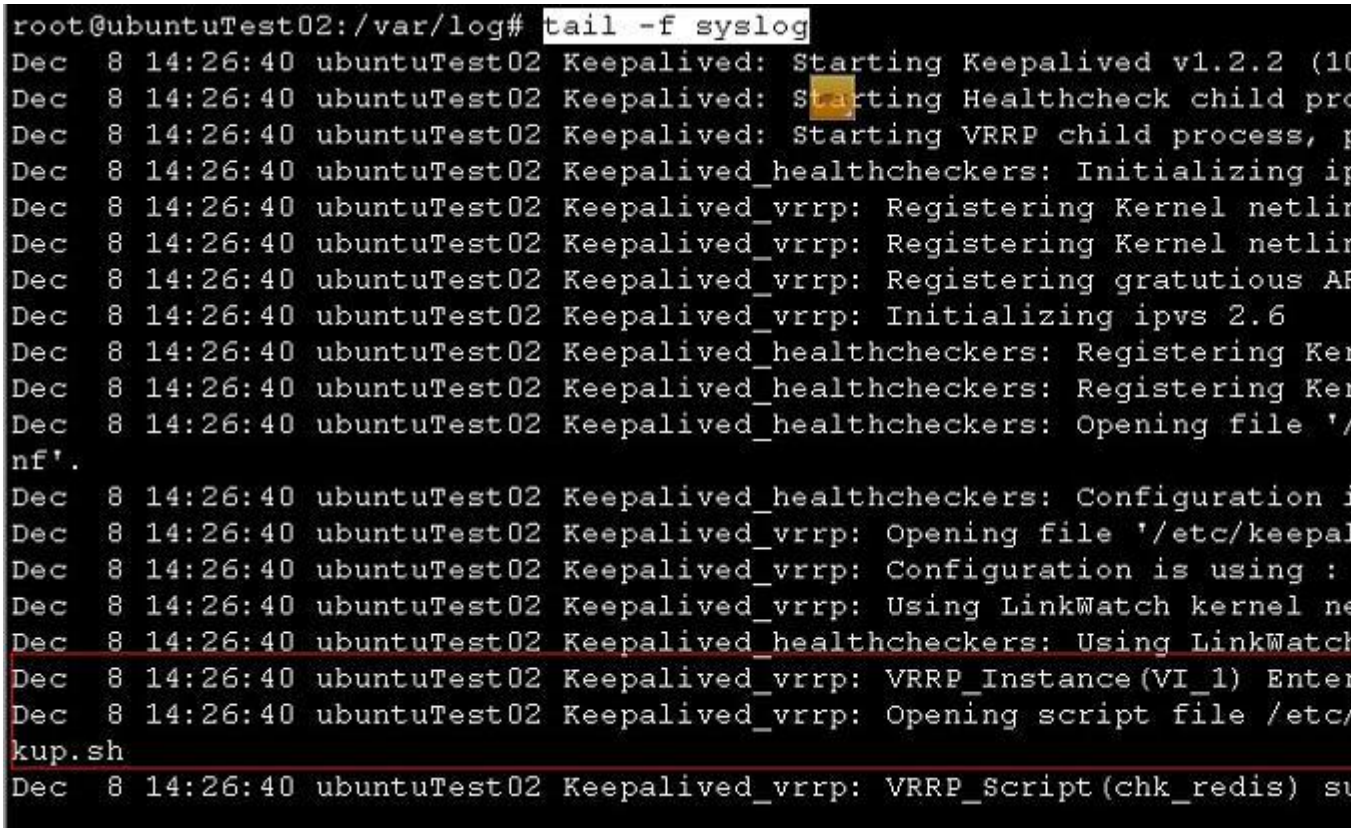
过程解释：

红框　确实如系统日志表现的那样，19 秒的时候先进入 backup 状态，执行 redis_backup.sh 脚本。redis_backup 是 keepalived 进入到 backup 状态时执行的脚本。进入到 banckup 状态，说明自身以前是 master 状态或者无状态，这时候会等待两秒种，让之前的 slave 把数据同步完，2 秒钟在数据大的时候有些短，其实是可以设置的，只要小于 3 倍的 advert_int 时间就行，因为 3 倍的时间之后会转为 master 状态的。　2 秒钟之后我们的 redis 连接到 27 上面(27 这时候充当 master)，开始进行正常的 master/slave 机制同步数据，也就是红框中 21 秒的时候。到此为止，redis_backup.sh 执行完毕。

绿框　由于设置了不抢占，所以过了 3 倍的 advert_int 时间之后，开始向 master 状态转变，这个转变过程需要的时间是不固定的，在上面的日志中需要 5 秒，正式的成为 master 状态，这时候会执行 redis_master.sh 脚本，脚本指定需要到 27 的 redis 上同步数据(27 在红框中的时候是充当 master 的)，你可以看到已经有连接了，那是因为在红框中已经连接过一次了，我设置的是同步 10 秒钟之后断开和 27 的连接，可以看到 49 秒的时候执行了 slaveof no one 命令，断开了和 27 的连接，把 26 自身的 redis 提升为 master，这时候数据也是最新的了。

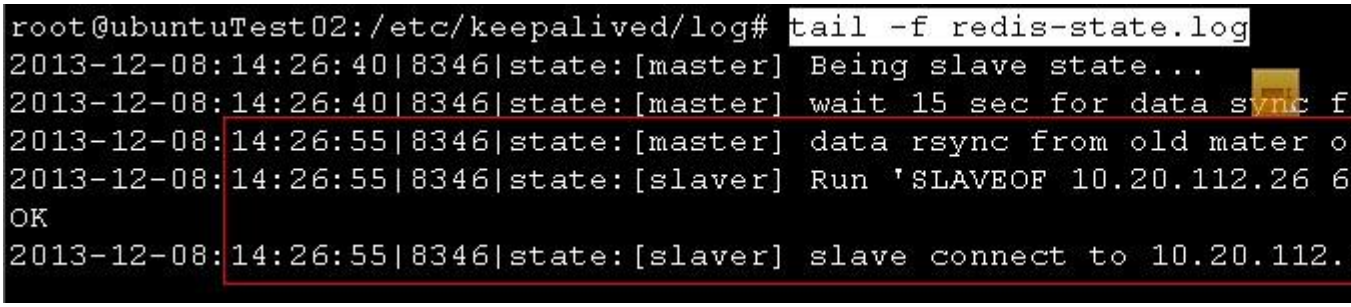
3 待 26 的各项进程都 ok 后，我们开始启动 27 上面的 keepalived 服务

27 的 syslog



红框 40 秒的时候启动 27 的 keepalived，由于默认都是 backup，所以直接进入 backup 状态，由于 27 的优先级比 26 低，所以 27 并不会过度到 master 状态，这时候 26 是 master 状态了。在 backup 状态执行了 redis_backup.sh 脚本。

27 的 redis-state.log



红框 在进入 backup 状态后，默认之前是 master 状态或者无状态，所以等待 15 秒，让 26 把数据同步完(26 这时候充当 backup)，在 15 秒之后，开始向 backup 转变，开始执行 slaveof 10.20.112.26 6379，作为 26 的 slave。

下面要模拟故障 3：

1 kill 掉 26 的 redis 进程，保持 keepalived 进程。(模拟 3)

26 的 syslog

```
Dec  8 14:25:39 ubuntuTest01 Keepalived_vrrp: VRRP_Instance(VI_1) E
Dec  8 14:25:39 ubuntuTest01 Keepalived_vrrp: Opening script file /
ter.sh



Dec  8 14:31:09 ubuntuTest01 Keepalived_vrrp: VRRP_Script(chk_redis
Dec  8 14:31:14 ubuntuTest01 Keepalived_vrrp: VRRP_Instance(VI_1) R
Dec  8 14:31:14 ubuntuTest01 Keepalived_vrrp: VRRP_Instance(VI_1) E
Dec  8 14:31:14 ubuntuTest01 Keepalived_vrrp: Opening script file /
kup.sh
```

红框　可以看到检测脚本 chk_redis 发现 redis 已经宕掉，降低了 26 的 keepalived 的优先级，导致 26 进入 backup 状态，开始执行 redis_backup.sh 监控脚本

26 的 redis-state.log

```
2013-12-08:14:25:49|2407|state:[slaver] data rsync from old mater o
2013-12-08:14:25:49|2407|state:[master] Run slaveof no one,close ma
OK
2013-12-08:14:25:49|2407|state:[master] wait other slave connect...




2013-12-08:14:31:14|3090|state:[master]
2013-12-08:14:31:14|3090|state:[master] Being slave state...
2013-12-08:14:31:14|3090|state:[master] wait 2 sec for data sync fr
2013-12-08:14:31:16|3090|state:[master] data rsync from old mater o
2013-12-08:14:31:16|3090|state:[slaver] Run 'SLAVEOF 10.20.112.27 6
Could not connect to Redis at 127.0.0.1:6379: Connection refused
2013-12-08:14:31:16|3090|state:[slaver] slave connect to 10.20.112.
```

红框　由于 redis 我是 kill 掉的，所以必然不能进行数据同步了，不过这并不影响使用。

27 的 syslog

```
Dec  8 14:26:40 ubuntuTest02 Keepalived_vrrp: Opening script file /
kup.sh
Dec  8 14:26:40 ubuntuTest02 Keepalived_vrrp: VRRP_Script(chk_redis




Dec  8 14:31:14 ubuntuTest02 Keepalived_vrrp: VRRP_Instance(VI_1) f
Dec  8 14:31:19 ubuntuTest02 Keepalived_vrrp: VRRP_Instance(VI_1) T
Dec  8 14:31:24 ubuntuTest02 Keepalived_vrrp: VRRP_Instance(VI_1) E
Dec  8 14:31:24 ubuntuTest02 Keepalived_vrrp: Opening script file /
ter.sh
```

绿框　可以看到 27 转变到 master 状态，并且执行了 redis_master.sh 脚本

27 的 redis-state.log

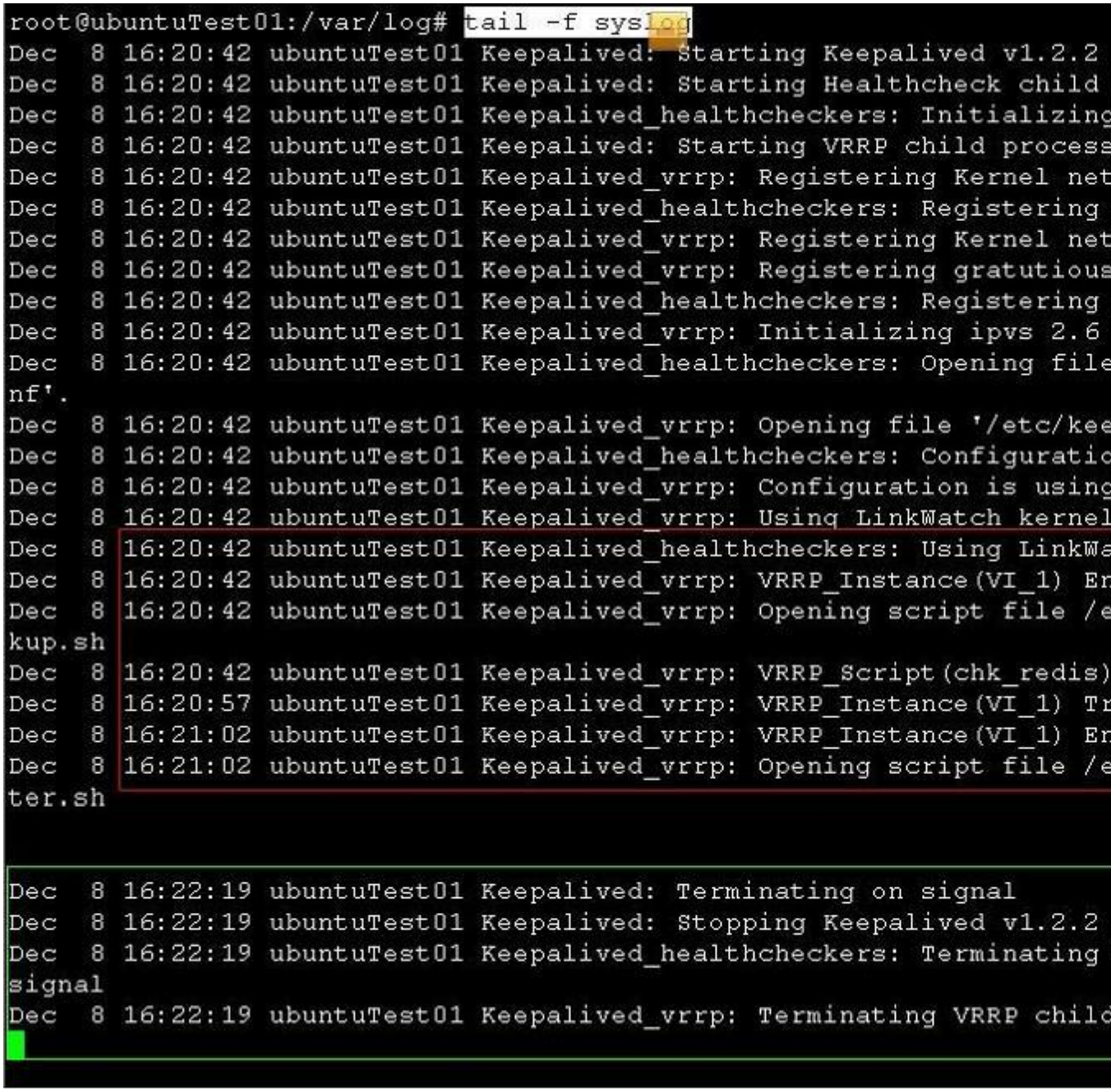
```
2013-12-08:14:26:55|8346|state:[slaver] slave connect to 10.20.112.




2013-12-08:14:31:24|8923|state:[backup]
2013-12-08:14:31:24|8923|state:[backup] Run 'SLAVEOF 10.20.112.26 6
OK Already connected to specified master
2013-12-08:14:31:24|8923|state:[backup] wait 10 sec for data sync f
2013-12-08:14:31:34|8923|state:[master] data rsync from old mater o
2013-12-08:14:31:34|8923|state:[master] Run slaveof no one,close ma
OK
2013-12-08:14:31:34|8923|state:[master] wait other slave connect...
```

绿框　由于 26 的 redis 是 kill 掉的,所以即使 27 转变到 master 状态也不能去 26 同步数据，当然 27 也等不到 26 连接上来，不过并不影响我们使用。

下面要模拟故障 2：

2 kill 掉 26 的 keepalived 进程，保持 redis 进程。(模拟 2)

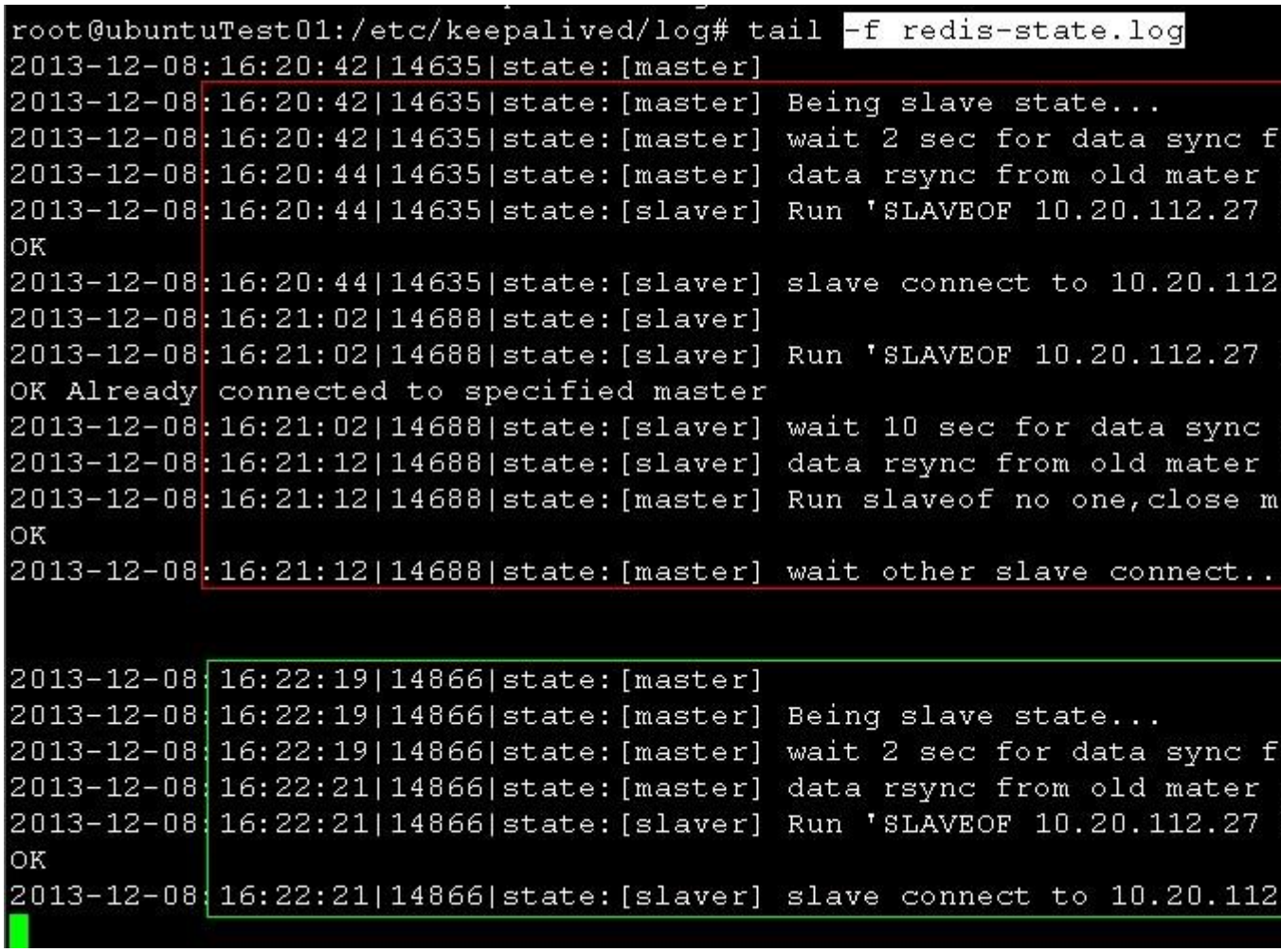
26 的 syslog



红框　是 keepslived 的启动过程，在模拟 3 中已经详细分析过

绿框　是 kill 掉 keepalived 进程后的日志。

26 的 redis-state.log



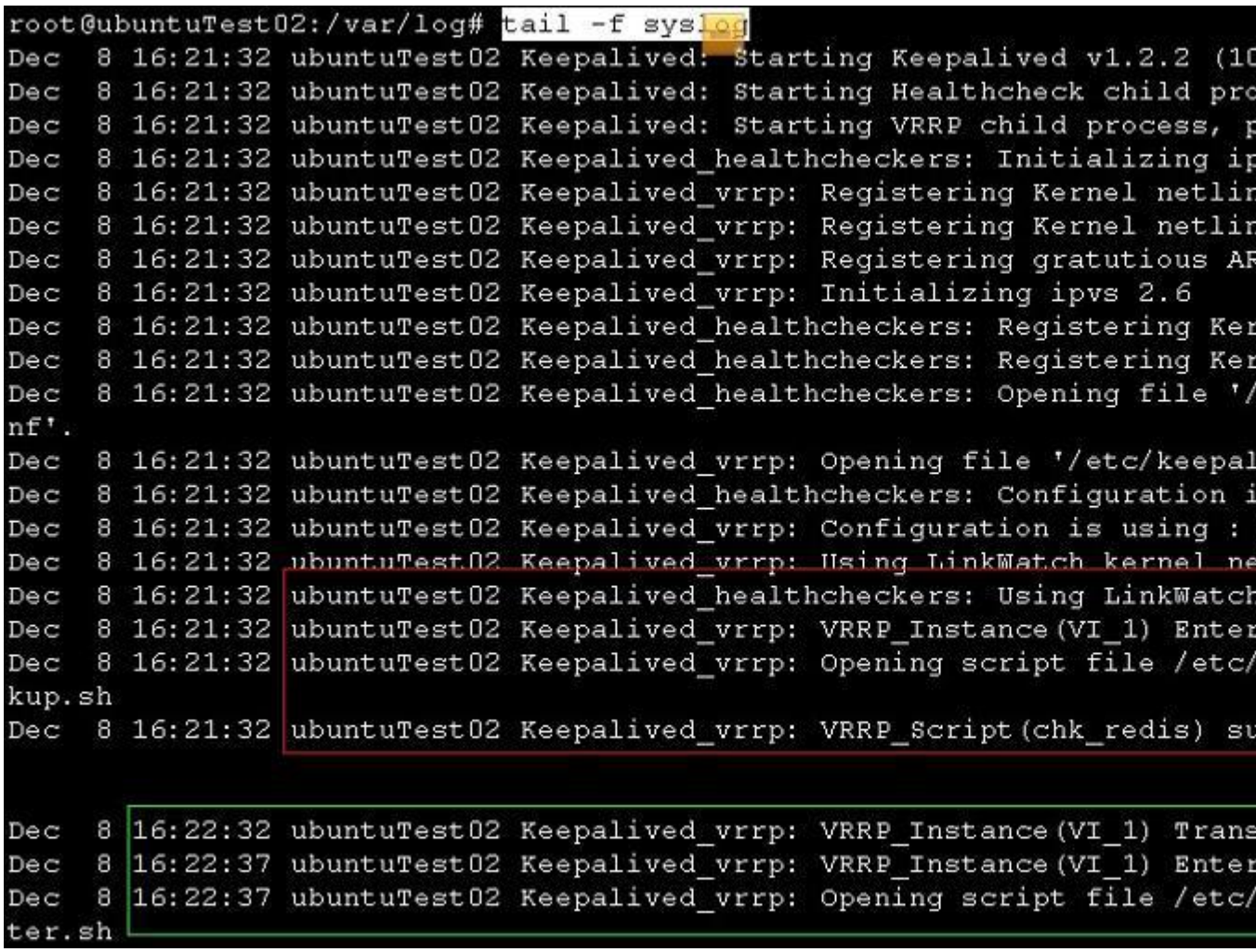
红框　是 keepalived 启动过程中执行的监控脚本日志，前面模拟 3 中已经详细分析过

绿框　是 keepalived 进程 kill 掉之后，执行的监控脚本日志，在等待 27 同步完数据后，作为 27 的 slave 连接上去。

26 的 redis info

可以看到 master/slave 的转变过程

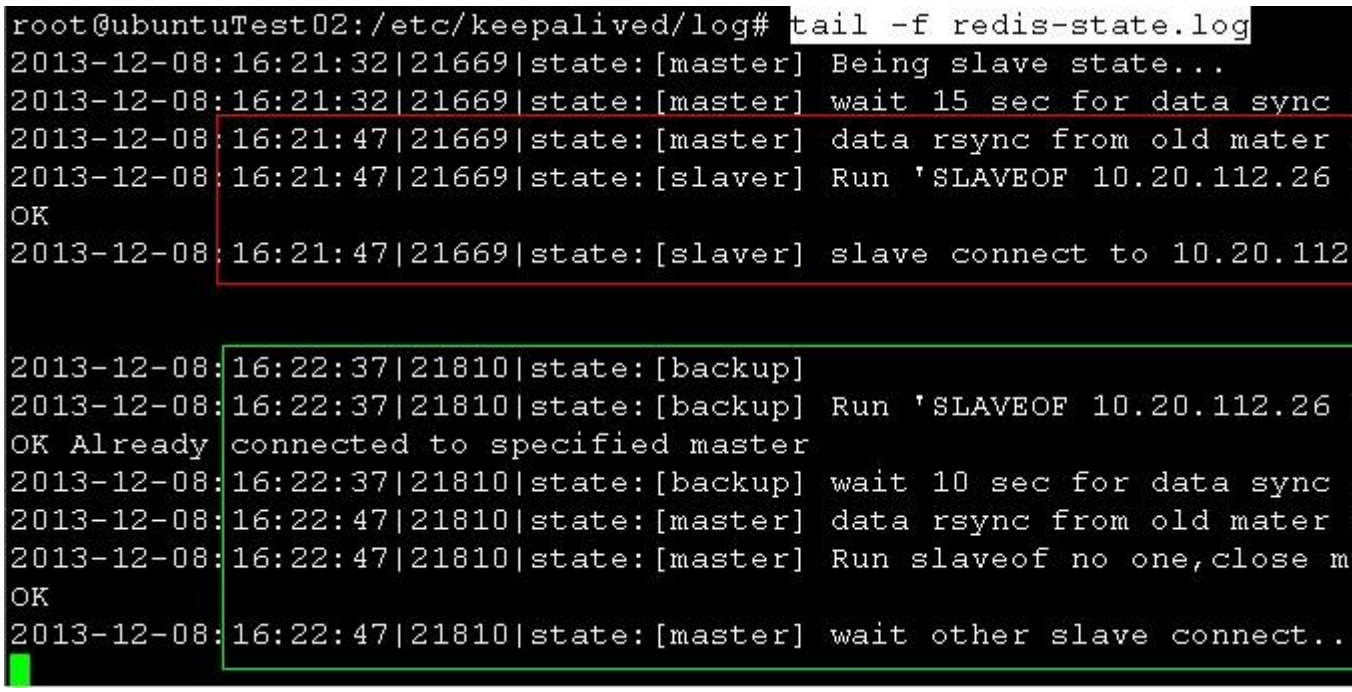
27 的 syslog



红框　是keepalived作为backup的启动过程,在模拟3中已经分析过。

绿框　是kill掉26的keepalived进程后，27转变为master时的日志，执行了redis_master.sh脚本。

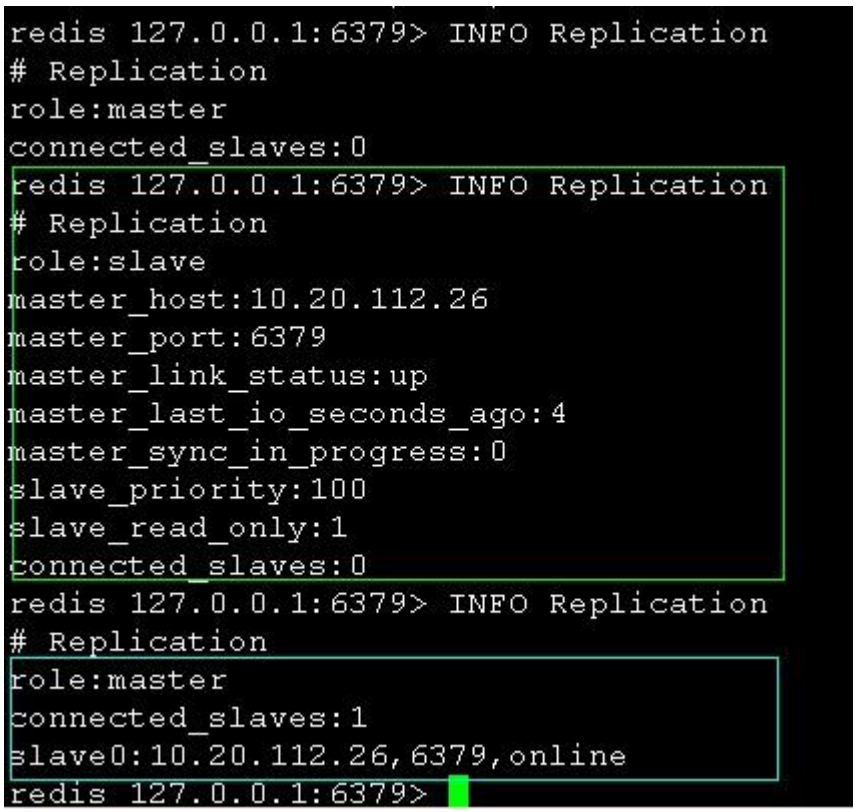
27的redis-state.log



红框　是 keepalived 启动时执行的监控脚本日志，由于开始是 backup 的，所以没有过多的状态切换过程。

绿框　是 26 的 keepalived 进程被 kill 掉之后，27 转变为 master 时执行的脚本日志，由于 keepalived 挂了，但是 26 的 redis 进程还在，所以先和 26 进行数据同步，完成之后再把自己提升为 redis master。

27 的 redis info



可以看到 redis 的 master/slave 变化过程

以上就是我所能想到的 keepalived+redis HA 方案中需要注意的地方。

原文
http://my.oschina.net/guol/blog/182491?utm_source=Tuicool_Weekly

Sparrow: 适用于细粒度 tasks 低延迟调度的去中心化无状态分布式调度器

背景介绍

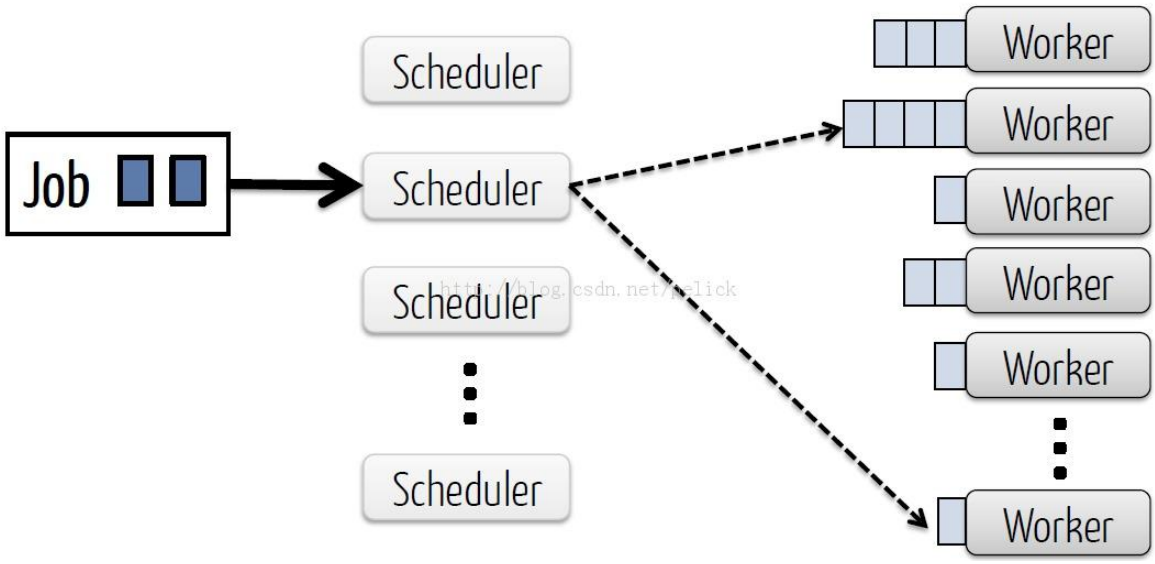
Sparrow 的论文收录在 SOSP 2013，在网上还可以找到一份作者的 talk ppt，值得一提的是作者是位 PPMM。她之前发表过一篇 The Case for Tiny Tasks in Compute Clusters，这篇文章我没有仔细看过，但是当时在看 mesos 粗细粒度模式的时候，组里有讨论过这篇论文。再结合她的 github 上的项目，发现她和 AMP 实验室里 Mesos，spark 等项目，在研究方向和合作上还是有很多渊源的。透过 Sparrow，结合对 Mesos、YARN 的一些了解，可以在理论和实际项目中获得更多一些关于资源调度的东西。

适用场景

Sparrow 是一个去中心化(分散)的无状态的分布式调度者，为细粒度 task 提供低延迟的调度。在一个由次秒级的 tasks 组成的 workload 里，调度器必须每秒为百万 tasks 提供毫秒级别延迟的调度决策，同时容忍调度失败。Sparrow 主要借助 **Batch Sampling + Late Binding + Constraints** 来达到较好的效果。下面会具体介绍 Batch Sampling 和 Late Binding，而 Constraints 是指用户可以对每个 job 进行一些约束和设定，比如所有 tasks 必须跑在有 GPU 的 worker 上，也就是在选择 worker 的时候增加一些条件约束。Constraints 可能会让用户在使用 Sparrow 的时候保持更高的好感，而真正的低延迟调度的提速应该还是取决于 Batch Sampling + Late Binding 的策略。

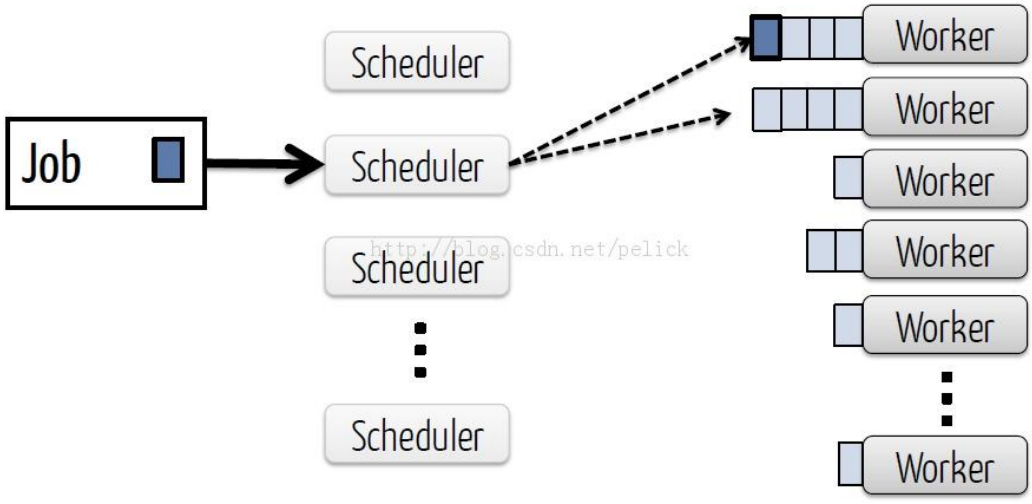
基础

sparrow 假设每个 worker 针对每个 framework 已经有一个 long-runing 的 executor 进程，最基础的是 random，将 Job 的 tasks 往随机选择的两个 worker 上分配，worker 自身维护了一个或多个 queue(当对用户产生隔离或对优先级有要求的时候会维护多个 queue)。



这种最基础的分配 task 的方法出自 The Power of Two Choices in Randomized Load Balancing，让调度器探测两个随机选择的 worker，把 task 分配到 queue 较短(仅考虑了已存在 tasks 数目)的 worker 上。Sparrow 基于 The Power of Two Choices in Randomized Load Balancing 的思想，提出了自己的三个改进的方法，并且给出了一些测试数据，说明了每种特性带来的效果提升。

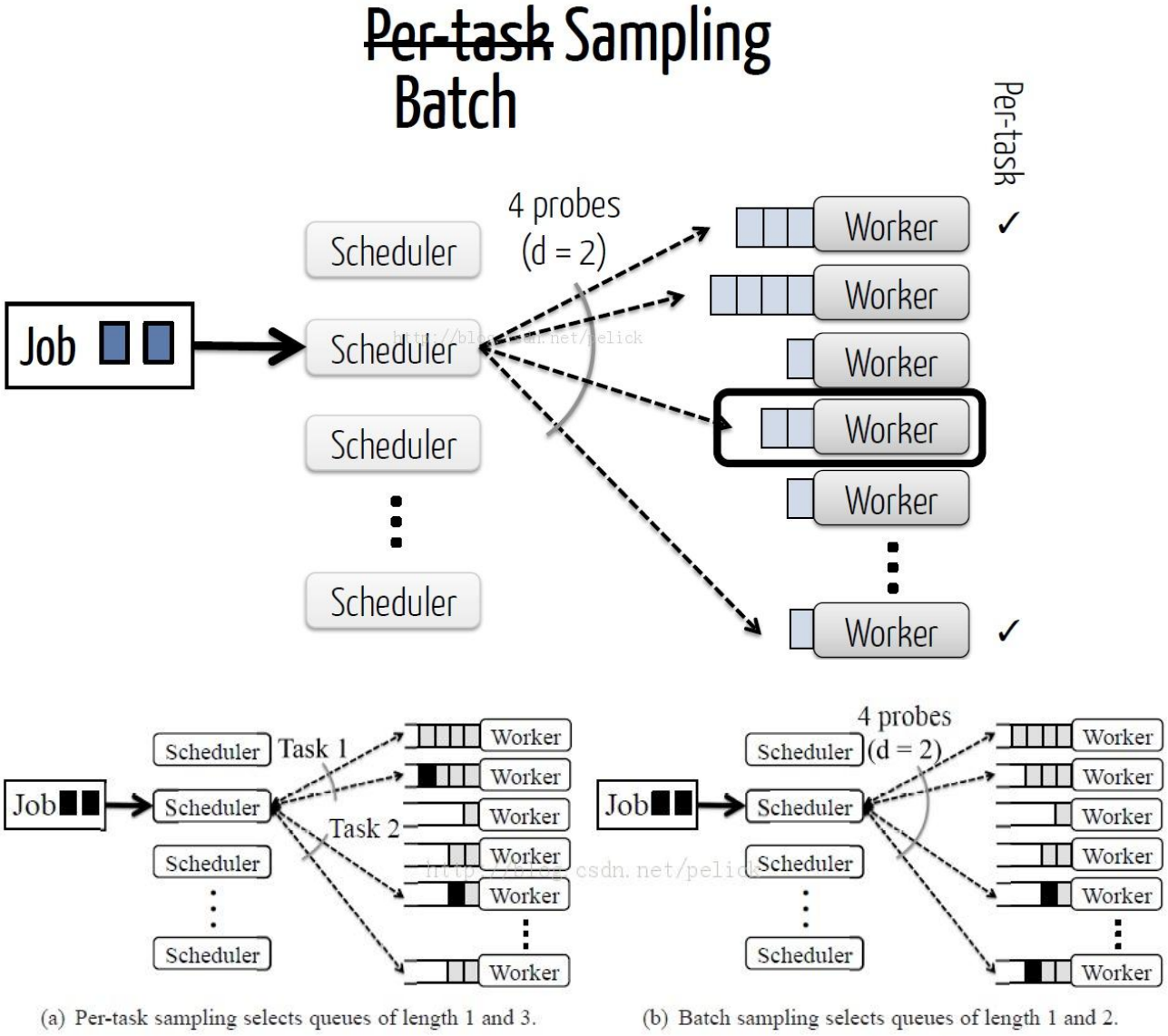
Sparrow 基于 random 的分配方式，进行的第一种改进是 Per-task sampling，即针对每个 task 都进行一次 random 操作，由 scheduler 发送 probe(探测器，一个轻量级 RPC)到 worker 上，然后选择一个队列比较短的 worker 分配 task。之后的 task 重新进行 random 的 work 选择。



Batch Sampling

上面的分配方式会让尾部的 tasks 等待较长的时间，Batch Sampling 改进了之前为 task 单独分配 random worker 的方式，让 scheduler 同时为一个 job 的 m 个 tasks 探测 m*d 个

workers(d>1)，为 tasks 一起监控很多个 worker。下图即 scheduler 为两个 task 同时探测四个 worker。批量取样的性能不会随着 job 并行化的增加而降级。

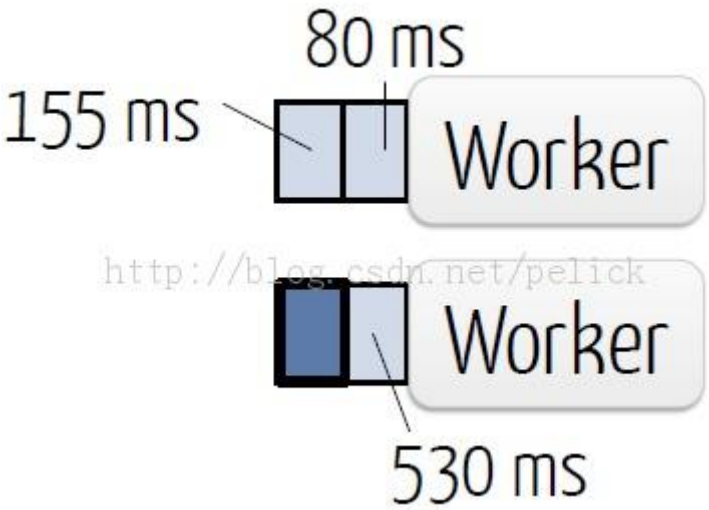


(a) Per-task sampling selects queues of length 1 and 3.

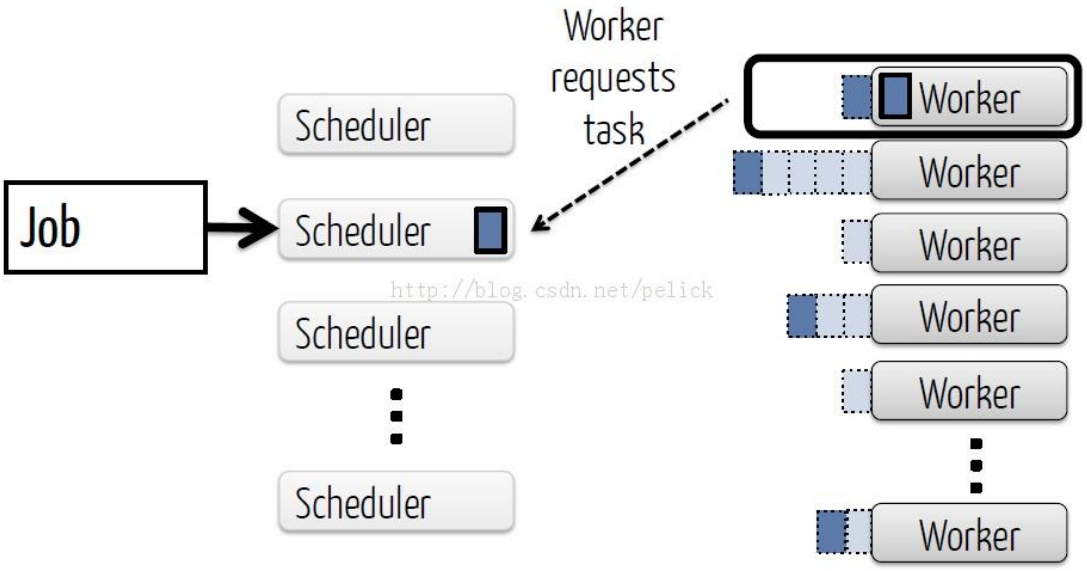
(b) Batch sampling selects queues of length 1 and 2.

Late Binding

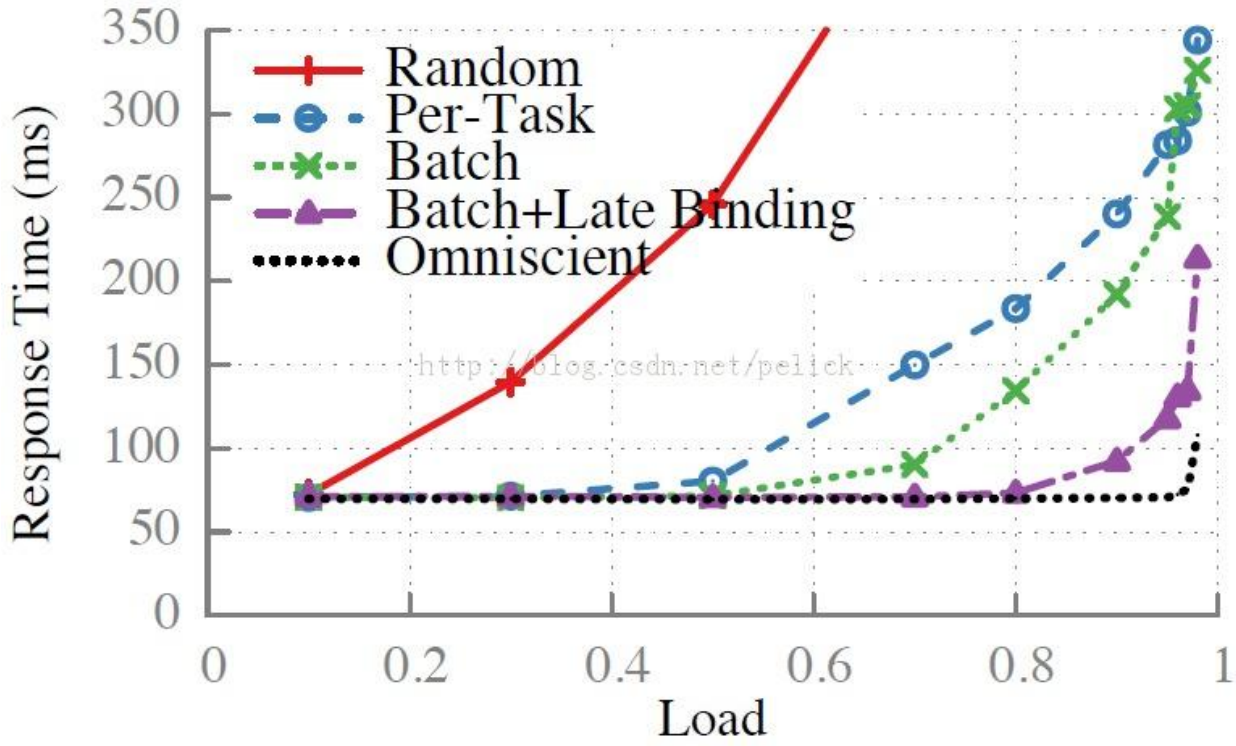
之前 random 模式里提到过，scheduler 对 worker 的选择是从已进行探测的备选 worker 里选择一个 queue 长度最短的 worker 放置新的 task，但是有可能造成较长的等待时间，因为没有考虑到已经排队的 tasks 的执行时间。如果 task 直接被分配在某个 worker 上排队，会造成较长的等待时间。



延迟绑定结合上面的批量采样的方式的话，就可以同时监控 d*m 个 worker，只要 worker queue 空了，就可以把 task 放置过去进行执行。下图中，承接上面的图示，Scheduler 选择了四个 worker 进行 batch sampling，当检测到第一个 worker 队列空了之后，就可以真正将 job 的两个 tasks 中的一个放到该 worker 上执行了，而另一个 task 继续由 scheduler 对四个 worker 进行监控。



在性能方面，下图展示了各个方法带来的延迟对比，其中黑色的虚线是假设调度器对于 worker 和 task 无所不知的情况下可以做出的最优调度，最靠近最优性能的是 batch + Late Binding 的曲线，从左往右对应的是上面的一个个图示，也证明了 Sparrow 的改进在百万级别细粒度 tasks 低延迟调度方面的进步。

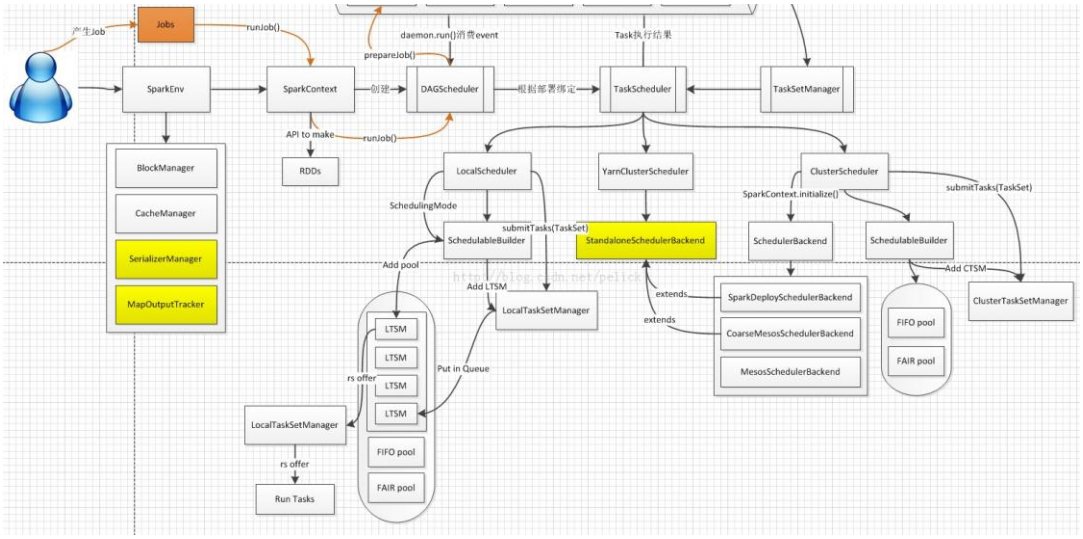


Sparrow & Spark

为了证明 Sparrow 的可用性，作者为 Spark 加了个 sparrow 调度插件，将 spark 的每个 stage 提交成为一个 Sparrow Job，提交给 sparrow 的 schedulers。但是 Sparrow 由于是无状态的，所以 framework 使用的 scheduler 如果 fail 了，需要自己检测到并且去连接备用 scheduler。在测试中 client 端获得一个 scheduler 列表，通过发送心跳的方式检测正在使用的 scheduler 有没有 fail，如果 fail 了，那应对措施也需要针对使用场景而设置，若放弃这次 job 重新在别的 sheduler 上启动重新分配 tasks 去执行的话，要保证幂等。Sparrow 同时也不会管 worker 的 fail 事件。项目分支地址。

我感觉 Sparrow 在地位上，比较像 Spark LocalScheduler 内的 LocalTaskSetManager，但是大于它，Sparrow 自己有一些 schedulers。Sparrow 假设每个 worker 针对每个 framework 已经有一个 long-runing 的 executor 进程，而这个 executor 可以是跑在 mesos 上的一个 executor process，也可以是 spark standalone 模式下在各个 slave 上起的 long-running 进程，Sparrow 做的事情是我给你一个 schedulers 列表，你用我的 scheduler 来处理你的 job 里的 tasks 怎么调度和分发，具体分发完执行的事情和 worker fail 了与 Sparrow 没有关系。下面图中，在 Spark Standalone 模式下，Spark 不借助于 Mesos 或者 YARN 这样的第三方资源调度系统，而是使用自己的调度模块。Spark 自己的调度模块里有 FIFO pool 和 FAIR pool，如果换成 Sparrow 的话，tasks 的调度不再是简单的先进先出(FAIR pool 内部仍然是先进先出的)，而应该是上面的 Batch sampling + lazy binding + constraints 的方式来调度到 slave 上。

(下图是我在阅读 Spark 源码时画的一张大图的一部分，右下角 ClusterScheduler 部分接的是 mesos 调度模块，在 SchedulerBackend 处可以接粗粒度或者细粒度的 mesos scheduler backend，左下方使用的是自己的调度模块 LocalScheduler，在 0.8 里除了之前的 FIFO pool 外，也新加入了 FAIR pool，在我之前的这篇博文里也介绍过。)



类似地，Sparrow 也实现了一个优先级队列，还实现了不同 user 之间的 queue 隔离性，这种情况下每个 worker 就维护了多个 queue。

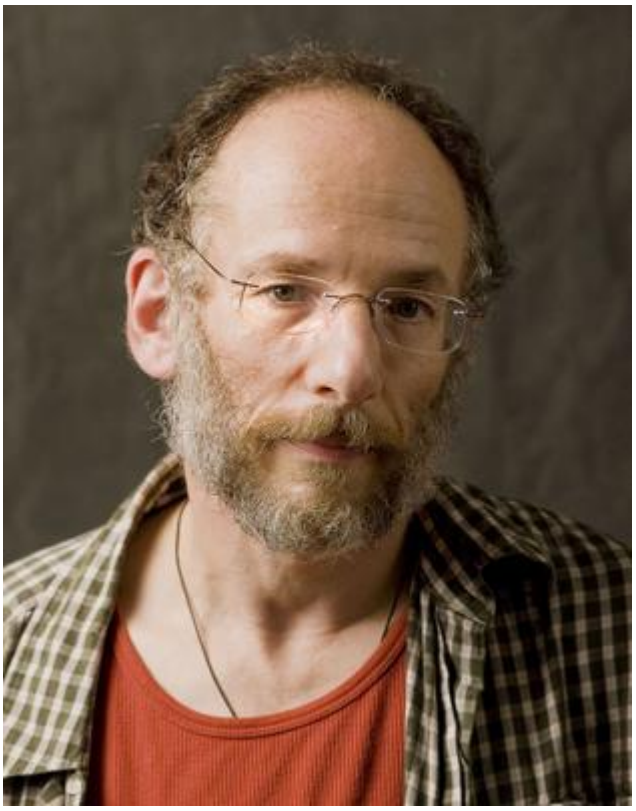
原文
http://blog.csdn.net/pelick/article/details/17093801?utm_source=Tuicool_Weekly

《C++ Primer》作者 Stanley B.Lippman 谈 C++语言和软件产业的发展

摘要：《C++ Primer》作者 Stanley B. Lippman 到访中国，并于 10 月 27 在由 CSDN

主办、电子工业出版社博文视点协办的 CSDN TUP 大师系列线下活动上做了主题演讲，介绍了 C++的新进展，本刊特约记者对 Lippman 进行了采访。

金秋 10 月，《C++ Primer》作者 Stanley B.Lippman 到访中国，并于 10 月 27 在由 CSDN 主办、电子工业出版社博文视点协办的 CSDN TUP 大师系列线下活动上做了主题演讲，介绍了 C++的新进展。活动间隙，本刊特约记者对 Lippman 进行了采访，形成此文。



勉励中国程序员要加强创新

《程序员》：您好，Lippman 先生，欢迎来到中国。我们上次见面，还是在 2004 年的微软 TechEd 期间，如今已经过去近 10 年了。您感觉中国软件行业和软件社区有哪些变化？

Lippman：事实上，我 2009 年还来过一次中国，那时在上海首次发表了现在称为 Hugo 的研究项目的主题演讲。

这些年，我所认识的大多数曾在微软等美国企业工作的中国人，现在都工作于中国国内企业了。而中国开发者社区则明显表现出了更强的自信心，或者说独立性。我是说，在 2004 年时显得更偏向于由西方主导，而现在更偏向于中国国内主导。

《程序员》：那么，您认为这种自信心源自哪里？

Lippman： 源自他们取得的成就。例如，你们已经有了自己的如同 Facebook 一般的社交网络工具，也有了自己的互联网。我 2009 年就去过中国的一家大型公司，深 刻感受到它做得非常成功。在那家公司里，我遇到不少曾在微软研究院的熟人，他们已经加入了那家公司了，而且他们显得信心满满，觉得自己就代表着未来。

《程序员》：您是否认为，现在中国的软件社区和是否越来越不受西方驱动了，并且拥有它们自己的想法和创新意识了？

Lippman： 我认为这还有待见证。这就好比自由软件基金会。光有设想还不能算是创新。自由软件基金会其实没做过太多有意思的事情，他们所做的大多数工作就是克隆现有的商业软件。而从目前看来，中国的软件社区做出来的有意思的事情也比较有限，它们也还是在创建一整套业界现有软件的仿制品。

但我期望看到的是，你们能够做一些创新的事情。目前而言，大多数的软件创新都还来自别处。你们要看一下自己的特色是什么。美国计算机行业现在变得相当软弱，人们不再勤奋 地工作。他们被惯坏了，只想快速赚钱，并且不再努力工作。中国的优势， 或者说中国程序员的优势在于，他们很勤奋地工作，并且渴望成功，愿意遵守纪律，也很投入工作。而在美国我看不到这种气氛了。

所以我认为，你们有着足够的能力在你们的国家做出伟大的事情，但这个时刻还不能说已经到来了。

《程序员》：我觉得您的说法是要鞭策我们不只是勤奋地工作，还要聪明地工作。

Lippman：就是这样。

回忆 C++语言诞生时期的峥嵘岁月，在贝尔实验室的黄金年代

《程序员》：您刚才说到，美国业界的目前情况是，人们不再努力工作。但我想，在 30 年前肯定不是这样。而您则经历过那个时代。我不知道您是否还记得，C++编 程语言最初的那些实现中，是否有一些完全由您实现出来的功能，如果有的话，您能否告诉我们您是如何设计它，以及它的哪些部分比较有意思呢？

Lippman： 我一向热衷于编译技术。当我开始与 Bjarne 合作工作于 Cfront 时，我在 1983 年从 Jerry Schwartz 那里了解到了 C++。当时 Jerry 工作于 Steve Johnson 的一个名叫 NAIL 的项目，即“新一代中间语言”，全部由 C++实现，而我当时则在从事 C++语言组的测试工作。然后，在 1985 年，Bjarne 决定发布 Cfront，即后来被称为“发行版 1.0”的实现。之所以这么急迫是因为，如果 C++老是处于实验阶段，而没有官方技术支持的 话，那么大学里的研究生就无法使用这门语言来做研究。我们为了让 C++能够成功发布而很努力地工作着，而我做的是测试工作。我问了一些很有难度的问题，他 对我做测试时的思路感到满意。

《程序员》：“他”是指 Bjarne 还是指 Schwartz？

Lippman：Bjarne， 他因为 Cfront 发行版 1.0 的工作而注意到了我，因为我总是询问有关该语言的问题。你知道，那时我们对 C++做了种种严苛的测试来发现 Bug。而我还 会问他一些微妙的问题，诸如某种特定代码的行为应该是怎样的，所以当我从测试小组转到编译器小组时，我可以选择做代码生成——这正是我感兴趣的——或者做 Cfront。我当时不太想做 Cfront，因为我对于前端没有兴趣，因为前端的逻辑很混乱，我是说，它没有一定的规则。而后端的工作，则有逻辑、有算法 得多。

我的朋友 Barbara E. Moo 当时是我的导师，我早已认识她多年，是她说服我接手 Cfront。实际上 Steve Dewhurst 也是我的好朋友，我知道你（高博）翻译过他的一本书《C++语言：99 个常见编程错误》。我在贝尔实验室的两位最要好的朋友就是 Barbara E. Moo 和 Steve Dewhurst。Steve 曾作为实习生与 Bjarne 一起工作过，因为他曾在贝尔实验室从事 C++编译器方面的工作。

当时他和 Bjarne 坐在同一间办公室里，办公室的大小与一般的宾馆房间差不多。他们并不总是相处得很好。当我开始做 Cfront 上的工作的时候，已经是 1986 年的 1.1 版的事情了。那时，有一个名叫 BlockingsField 的以色列公司，他们的一些员工当时正把 Cfront 移植到 PC 平台上，那个年代的 PC 使用的是 16 位段式体系结构，这是因为地址只有 16 位宽的缘故，而在 Unix 上我们使用的是 32 位地址。我们都不希望与 PC 打交道，因为都觉得它是垃圾——从计算机科学的角度来看的话。但我们别无选择，必须发布一个 PC 版本，于是我们只得做一些事情来使它变为可能，比如，添加了无符号整数。我是说，在 1.1 版发布之后我们又不得不添加了一些东西。

《程序员》：就是说，C++中的无符号整数是您添加的？

Lippman：Cfront 的最初版本中是没有无符号整数的，Cfront 是 Bjarne 的一个研究项目，所以它当时并没有像一个正式发布版的那种产品级成熟度。例如，如果你使用 Cfront，而我们还没有实现某种特性，我们可以让它显示“对不起，还没有实现”或者我们会说“如果你这样做的话，会有小鬼缠身哟”。你知道，我们在正 式发布版里面不可能这么干，但我们一开始做的并不是正式发布版。因为最初的 Cfront 只要 50 美元就能获得源代码，几乎就相当于免费了，所以任何人都能 购买 C++完整的源代码。

Bjarne 知道，我可以与同事们和睦相处，因为我当时比较年轻，而我又不很懂计算方面的东西，所以我从不与他争 论，因为他懂的东西是我所知的上百倍！在 1.1 版以后，他作为研究者，继续他有关多重继承特性的工作，而我是唯一的 Cfront 支持人员，也是唯一的开发人员。所以基本上大家所知的关于 C++的每个特性：成员函数、指向成员函数的指针等等，都是我实际地把它们一个个添加到 Cfront 中去的。所以那是一段 对我而言非常激动人心的经历。我和 Bjarne 相处得非常融洽，因为我不喜欢问问题——我喜欢自己研究出问题的答案，而他也不太喜欢回答问题。所以他不来 管我，我也不打扰他，大家反而相处得很好。

C++语言从实验室步入成熟所经历的阵痛

《程序员》：您是说，在早期的 C++语言实现中，像指向成员函数的指针这样的特性，都不需要通过一个非常正式的评审过程就能加入吗？

Lippman： 当时完全不具正式性。拿 Cfront 发行版 1.0 来说，我测试了它，写了所有的文档，参与了它的移植。所以当时我像是 Cfront 的单人乐队，因为在产品 部门内部没有任何支持，也还没有人需要这种支持。当时所有人都说：我要用 Fortran 77，或者我要用 Ada，没有人关心 C++。

Steve Johnson 被调离以后，新上任的部门经理 Carlson Rillo 完全不了解计算机语言，所以他所知道的所有事情就是，他需要 Ada 以获得认证，从而获得政府部门的合同；他需要 Fortran 77，因为有一些公司想要买。但他不知道，也不关心 C++。事实上，他就是当初那个问我老板 Barbara 奇怪问题的家伙：“为什么 Lippman 在写一 本关于 C++的书？他到底是什么来头？”

我们都看不起这家伙，他是个打高尔夫的——此人不过是被强行委派到我们小组头的人物，并无远见。只 是因为 Steve 当初管的事儿范围太大了一点，所以才失控了，但在我眼中，Steve 完全称得上是一个研究员，而 Carlson Rillo 则是一个彻头彻尾的职业经理人。不管怎样，总地来说那段时间是非常精彩的，是我职业生涯中最美好的日子。

《程序员》：随后在迪斯尼和微软的经历如何？

Lippman：我在迪斯尼工作的那段时间可就没那么美好了，你也看得出来，我在一个高度组织化的环境中工作得并不顺心，那里有很多条条框框。所以，当我一开始工作于 Cfront 上时，那里并没有规则，只有我和 Bjarne。

刚来到迪斯尼时，我是首席软件工程师，还有一位是 Peter De Vroede，我们一起参与软件工程，并且是那里仅有的来自计算机科学背景的软件工程师，而其他每个人都是动画师、动画制作人员，像 Scott Johnson、M. J. Turner 等等。他们都是非常聪明的人，但他们不是工程师，所以他们写出来的代码很不修边幅，而我在那种环境中能发挥出我的能力。但随着环境变得越来越结构化，就很难再有机会做些什么事情出来。当所有东西都变成了委员会机制时，我在那样的环境中就不能出色地工作了。

这也就是为什么我在微软干得并不出色，因为在微软，程序员不能决定任何东西，能作决定的是项目经理（Program Manager），他们是接触用户的那些人。但用户并不知道他们想要什么，他们只知道他们现在有什么、现在喜欢什么。例如，对于 Cfront，我可能接到 一个开发人员打来的电话，他说他遇到了一个 Bug，我就会修复它，并发送一个补丁给他。我不需要通过什么审批，他们马上就能得到这个补丁。但在微软，可能 需要等上两年才能得到一个补丁。因为它必须通过委员会的审批，所以必须等待相当长的时间才能被发布。并且 Oracle 这样的企业用户，虽然需要等待，但你 还是能得到一个补丁。可是如果你只是一个大众开发者，你将被彻底无视。所以我对你问题的回答就是，我们一路过来基本上都是自己做的，直到标准委员会成立为 止。

标准委员会被发展起来以后，生活就不再那么愉快了。例如，这边我和 Bjarne 想做一个特性，我就会试着实现它，看看它工作起来效果如何。但下一次会议时，标准委员会说，不，我们要收回这个特性。于是就不那么愉快了。

《程序员》：确实是这样，当组织越来越大，用户群体爆炸式增长以后，组织里面就有了越来越多的决策者。

Lippman： 这就变得相当无趣。例如，在微软，所有的东西都必须达成共识。这样做的问题在于，微软的很多人带着他们的笔记本电脑参加会议，他们在会上首次听说某个设想然后就发表观点，而他们之前都从来没有听过这个设想。但你知道，对于计算机问题，你必须思考得足够深入，因为它们总会产生种种后果，所以这样不经过深思熟 虑是非常令人沮丧的。

我当初的职责是尝试找到把 C++移植到.NET 上去的方法。我去洛杉矶，在这个问题上工作了个把月。然后我向他们提出 并演示给他们看。接着，有人会说：“不，这样不好”，但他们之前连听都没听到过这些思想。还有人会说“那样做的用处是什么啊？”所以你问我之前负责哪方面 工作，我的回答是负责所有方面的工作：Bjarne 发明了那些特性，我负责实现它们。

而有意思的是，因为你实现了它们，而当时又没有任何标 准，就很少有人了解那些特性是什么，除非通过编译器来检验。所以，大概四年来，我自己实现 GCC 的 Michael Tiemann、实现 Oracle C++编译器的 Michael Bob、实现 PC 上编译器的 Walter Bright，我们是当时仅有的几个真正理解这门语言的人，当然，还包括 Bjarne。你知道，这也就是为什么人们想要建立语言标准的原因。

但问题是，一旦你有了标准以后，所有事情都变得困难起来。因为，比如说，微软想要异常处理的另一种模型，然后 Bjarne 觉得技术上这是正确的。他们想要一 种恢复语义的执行期模型，这是出于他们自己的考虑，于是事情就变得政治化了。然后整个过程就变成了如何获得你的支持者联盟了，而且，突然间，Bjarne 无法思考并做出决策了——他必须在利益各不相同的多个集团之间取得共识，比如，公司有自己的利益，使得我们无法在二进制兼容性上达成一致。

再例如，微软有一个虚拟基类指针实现，实在拙劣得很。但微软为此申请了专利，于是造成巨头之间爆发了标准战争，兼容性就被抛在一边了，这也是为什么微软要发 明 COM 来解决这个问题。因为如果你无法在编译器之间形成二进制兼容性，就意味着人们就必须用 PC 上的各个编译器都发布一个库，而这几乎是不可能的。

这个工作量太大了，对吧？如果你想要发布一个字符串库，你必须在 Borland 上实现一遍，又在 Zortec 上实现一遍，又在微软编译器上实现一遍。然后你 又必须在 Sun 上面实现一遍……因此，在我看来，它已经破坏了 C++本身所拥有的优秀特性。所以它给我留下了糟糕的印象。

《程序员》：这听起来与 C++设计哲学背道而驰。

Lippman： 是啊，但为了成为一个业界的标杆，标准还是需要的，只是这方面的工作已经不再有趣了。这也就是为什么当有机会和 Bjarne 一起去研究部门做 GRAIL 项 目时，我就抓住这个机会调过去了，因为这是我们一拨人能做令人兴奋的事情的又一次机会。做那个项目的时候就是《深度探索 C++对象模型》一书问世的时候 ——我当时的工作就是建立那个模型。

《程序员》：如果我没有记错的话，最后 Bjarne 决定不采纳恢复语义，而转向了中止语义？

Lippman：但这是因为他没有研究过现有的模型。他咨询了好多实现者，求证他们的意见，他已经尽力而为了。

谈天才与后天努力，Lippman 眼中最好的 C++编译器

《程序员》：您在中国有很多粉丝，包括我在内，都认为您是一个真正的天才。

Lippman：我不是天才。我很幸运，而且我工作得很勤奋。但我不是一个天才。

《程序员》：那么我们能不能说，你在正确的方向上作出了努力？

Lippman： 有一本书叫做《权力的游戏》，它也被改编成了一个 HBO 电视连续剧，或者说是魔幻剧。里面有一个侏儒角色名叫 Tyrion，他总是不停地读书。他的哥哥是 一个勇士，一个帅哥；而他的爸爸也是一个勇士，有权有势。所以有人问他为什么读了那么多书。他回答说：“孩子，因为我是一个侏儒。如果我没有智慧，那就没有人会尊重我了。”所以我努力工作因为我必须这样做，而不是别

的原因。因为我来自蓝领阶层，来自主流文化之外，没有读过好的学校。我的父母都是外来移民。 除了获取成就，我别无他途。

《程序员》：您曾参与第一个 C++编译器 Cfront 的开发，但您对它的实现并不满意，能谈谈其中的原因吗？

Lippman：我想实现一个特性，他们答应我让我来实现异常处理。但组织里的某些人不认为我有能力完成它。所以他们把这项任务转交给惠普。惠普后来交付给我们一个 Cfront 版本，在这个版本上，一个带异常处理的 Hello World 花了 15 分钟才编译完成。

《程序员》：听起来很不可思议，这样的产品性能是无法接受的。

Lippman： 很遗憾，Cfront 以这种方式终结了。如果我来做的话，无论如何不可能做得比这还糟糕，对吧？而这也导致了后来 EDG 编译器的出现，因为 John Spicer 和我曾在同一家公司工作，而当 Cfront 失败以后，他加入了 EDG，他们的编译器就成了业界标杆。它是当时最优秀的 C++编译器。而现在最优秀的则是 LLVM，它是一个出自大学的编译器，Google 和苹果都对它作了不少投资，但它仍是一个出色的编译器。它也支持 Objective-C，和 Xcode——苹果的编译系统。我认为它是市面上最出色的编译系统。

《程序员》：原来如此，也许使用 LLVM 来测试一下 C++11 新标准会是有价值的尝试。您对 C++最初的标准有何评价？

Lippman： 好的，结果肯定会很有趣。我想说，C++标准的第一版，是一次很糟糕的失败。他们把数学库放进去了，接下去他们又后悔，想要说服人们不要使用它们。异常处理也工作得不理想，constraint 子句无法被真正地用在任何大型编程项目中，用于确定实例化的 Koenig 查找规则无法正常工作。

换句话说，C++的第一版标准中有很大一部分无法正常工作。他们后来花费了巨大的努力，才修复了大多数的问题。但我想强调一点，C++在当时还只是在一个很小的社区里流行，其中有些人不惜代价而投身于此，最终成为了真正的专家。

原文
http://www.csdn.net/article/2013-11-27/2817632?utm_source=Tuicool_Weekly

趣味横生的程序员搞怪代码注释

相信每一个编程极客都知道什么是注释，也都知道如何在代码中添加注释，今天这篇文章中，我们将不会讨论如何添加注释，或者如何添加一个完美的注释，在今天的文章里，我们将给大家奉献

相信每一个编程极客都知道什么是注释，也都知道如何在代码中添加注释，今天这篇文章中，我们将不会讨论如何添加注释，或者如何添加一个完美的注释，在今天的文章里，我们将给大家奉献一场来自全球开发人员的注释盛宴，看看大家是怎么在代码中添加自己富有想象力的注释吧，绝对会让你乐此不彼！

当然，如果你也有很多超有趣的注释，请留言和我们分享！我们的口号是：“快乐编程，娱乐注释” !!!

注重语法的注释

```
return 1; # returns 1
```

来自绝对菜鸟的注释

```
// I am not sure if we need this, but too scared to delete.
...
...
```

中文：个人不确认是不是需要，但是实在不敢删除

来自正直程序员的注释

```
// I am not responsible of this code.
// They made me write it, against my will.
```

中文：个人不负责这块的质量，因为他们逼迫我违心的写了这段代码

来自电影的注释

```
options.BatchSize = 300; //Madness? THIS IS SPARTA!
```

中文：疯了吧？这是斯巴达！

绝对尽责的注释

```
i++; // increase i by 1
```

中文：给变量 i 增加一个记数

绝对会被忽略的注释

```
Catch (Exception e) {
  //who cares?
}
```

中文：谁在意？

绝对不能信任注释

```
/**
  * Always returns true.
  */
public boolean isAvailable() {
        return false;
}
```

中文：返回为 true （编辑：永远不能相信注释）

典型的遗留系统代码里的注释

```
/*
  * You may think you know what the following cod
e does.
  * But you dont. Trust me.
```

```
4.     * Fiddle with it, and youll spend many a sleeples
   s
5.     * night cursing the moment you thought youd be cle
   ver
6.     * enough to "optimize" the code below.
7.     * Now close this file and go play with somethin
   g else.
8.     */
```

中文：你可能相信你能看懂以下代码，但是其实绝对不可能，相信我。一旦你调试了，你绝对会后悔装聪明去尝试优化这段代码。最好的方式是关闭文件，去玩点儿你喜欢的东西吧。

Java 程序里经常能看到的“典型”注释

```
1.   try {
2.
3.   } finally { // should never happen
4.
5.   }
```

中文：绝对不会运行到这里

超级有自知之明的代码注释

```
1.   //This code sucks, you know it and I know it.
2.   //Move on and call me an idiot later.
```

中文：这段代码的确很挫，我知道你也知道，先不要骂我 2B 了，请先继续往下看

绝对有年头的注释

```
1.   long long ago; /* in a galaxy far far away */
```

中文：在很远很远的银河系外 （编辑：这段代码能运行，绝对是个奇迹）

"情色"代码让我如何注释为好

```
1.   double penetration; // ouch
```

中文：我擦！（编辑：如果你不熟悉英文，double penetration 绝对无法理解，但是如果你熟悉 AV，了解情色，应该知道什么体位是“双管齐下”吧！ 嘿嘿，有点儿淘气了，请大家不要见怪）

绝对无法挑剔的注释

```
1.  ////////////////////////////////////  this  is  a  well  c
    ommented  line
```

中文：这注释绝对完全没有问题

保证正确体位的注释

```
1.  //  I  don't  know  why  I  need  this,  but  it  stops  th
    e  people  being  upside-down
2.  x  =  -x;
```

中文：我也不知道为什么需要这个，但是这个能保持大家不会倒立

来自 Java1.2 SwingUtilities 的注释

```
1.  doRun.run();    //  ...  "a  doo  run  run".
```

最好的帮助你了解递归的注释

```
1.  #  To  understand  recursion,  see  the  bottom  of  this  fi
    le
2.
3.  At  the  bottom  of  the  file:
4.
5.  #  To  understand  recursion,  see  the  top  of  this  fil
    e
```

中文：#如果想了解递归，请看最下面的注释...如果想了解递归，请看最上面的注释

绝对督促你工作的注释

```
1.  /*  Please  work  */
```

绝对菜鸟注释

```
1.  //I am not sure why this works but it fixes the problem.
```

中文：不知道为什么，但是的确解决了这个问题。
原文：http://www.cocoachina.com/gamedev/misc/2013/1205/7478.html?utm_source=Tuicool_Weekly